Introduction to Programming Using Java

Version 6.0, June 2011

(Version 6.0.3, with minor corrections, January 2014)

David J. Eck

Hobart and William Smith Colleges

David J. Eck (eck@hws.edu)
Department of Mathematics and Computer Science
Hobart and William Smith Colleges
Geneva, NY 14456

The web site for this book is: http://math.hws.edu/javanotes

Contents

Preface

Introduction to Programming Using Java is a free introductory computer programming textbook that uses Java as the language of instruction. It is suitable for use in an introductory programming course and for people who are trying to learn programming on their own. There are no prerequisites beyond a general familiarity with the ideas of computers and programs. There is enough material for a full year of college-level programming. Chapters 1 through 7 can be used as a textbook in a one-semester college-level course or in a year-long high school course. The remaining chapters can be covered in a second course.

The Sixth Edition of the book covers "Java 5.0", along with a few features that were introduced in Java 6 and Java 7. While Java 5.0 introduced major new features that need to be covered in an introductory programming course, Java 6 and Java 7 did not. Whenever the text covers a feature that was not present in Java 5.0, that fact is explicitly noted. Note that Java applets appear throughout the pages of the on-line version of this book. Most of the applets require Java 5.0 or higher.

The home web site for this book is http://math.hws.edu/javanotes/. The page at that address contains links for downloading a copy of the web site and for downloading PDF versions of the book.

* * *

In style, this is a textbook rather than a tutorial. That is, it concentrates on explaining concepts rather than giving step-by-step how-to-do-it guides. I have tried to use a conversational writing style that might be closer to classroom lecture than to a typical textbook. You'll find programming exercises at the end of each chapter, except for Chapter 1. For each exercise, there is a web page that gives a detailed solution for that exercise, with the sort of discussion that I would give if I presented the solution in class. (Solutions to the exercises can be found only in the web version of the textbook.) I **strongly** advise that you read the exercise solutions if you want to get the most out of this book.

This is certainly not a Java reference book, and it is not a comprehensive survey of all the features of Java. It is **not** written as a quick introduction to Java for people who already know another programming language. Instead, it is directed mainly towards people who are learning programming for the first time, and it is as much about general programming concepts as it is about Java in particular. I believe that *Introduction to Programming using Java* is fully competitive with the conventionally published, printed programming textbooks that are available on the market. (Well, all right, I'll confess that I think it's better.)

There are several approaches to teaching Java. One approach uses graphical user interface programming from the very beginning. Some people believe that object oriented programming should also be emphasized from the very beginning. This is **not** the approach that I take. The approach that I favor starts with the more basic building blocks of programming and builds from there. After an introductory chapter, I cover procedural programming in Chapters 2, 3, and 4. Object-oriented programming is introduced in Chapter 5. Chapter 6 covers the closely

related topic of event-oriented programming and graphical user interfaces. Arrays are covered in Chapter 7. Chapter 8 is a short chapter that marks a turning point in the book, moving beyond the fundamental ideas of programming to cover more advanced topics. Chapter 8 is about writing robust, correct, and efficient programs. Chapters 9 and 10 cover recursion and data structures, including the Java Collection Framework. Chapter 11 is about files and networking. Chapter 12 covers threads and parallel processing. Finally, Chapter 13 returns to the topic of graphical user interface programming to cover some of Java's more advanced capabilities.

* * *

Major changes were made for the **previous** (fifth) edition of this book. Perhaps the most significant change was the use of parameterized types in the chapter on generic programming. Parameterized types—Java's version of templates—were the most eagerly anticipated new feature in Java 5.0. Other new features in Java 5.0 were also introduced in the fifth edition, including enumerated types, formatted output, the *Scanner* class, and variable arity methods. In addition, Javadoc comments were covered for the first time.

The changes in this **sixth** edition are much smaller. The major change is a new chapter on threads, Chapter 12. Material about threads from the previous edition has been moved to this chapter, and a good deal of new material has been added. Other changes include some coverage of features added to Java in versions 6 and 7 and the inclusion of a glossary. There are also smaller changes throughout the book.

* * *

The latest complete edition of *Introduction to Programming using Java* is always available on line at http://math.hws.edu/javanotes/. The first version of the book was written in 1996, and there have been several editions since then. All editions are archived at the following Web addresses:

- First edition: http://math.hws.edu/eck/cs124/javanotes1/ (Covers Java 1.0.)
- Second edition: http://math.hws.edu/eck/cs124/javanotes2/ (Covers Java 1.1.)
- Third edition: http://math.hws.edu/eck/cs124/javanotes3/ (Covers Java 1.1.)
- Fourth edition: http://math.hws.edu/eck/cs124/javanotes4/ (Covers Java 1.4.)
- Fifth edition: http://math.hws.edu/eck/cs124/javanotes5/ (Covers Java 5.0.)
- Sixth edition: http://math.hws.edu/eck/cs124/javanotes6/ (Covers Java 5.0 and later.)

While it is not actually required by the license, I do appreciate hearing from people who are using or distributing my work.

* * *

A technical note on production: The on-line and PDF versions of this book are created from a single source, which is written largely in XML. To produce the PDF version, the XML is processed into a form that can be used by the TeX typesetting program. In addition to XML files, the source includes DTDs, XSLT transformations, Java source code files, image files, a TeX macro file, and a couple of scripts that are used in processing.

I have made the complete source files available for download at the following address:

http://math.hws.edu/eck/cs124/downloads/javanotes6-full-source.zip

These files were not originally meant for publication, and therefore are not very cleanly written. Furthermore, it requires a fair amount of expertise to use them effectively. However, I have had several requests for the sources and have made them available on an "as-is" basis. For more information about the source and how they are used see the README file from the source download.

* * *

Professor David J. Eck
Department of Mathematics and Computer Science
Hobart and William Smith Colleges
300 Pulteney Street
Geneva, New York 14456, USA
Email: eck@hws.edu
WWW: http://math.hws.edu/eck/

Chapter 1

Overview: The Mental Landscape

When you begin a journey, it's a good idea to have a mental map of the terrain you'll be passing through. The same is true for an intellectual journey, such as learning to write computer programs. In this case, you'll need to know the basics of what computers are and how they work. You'll want to have some idea of what a computer program is and how one is created. Since you will be writing programs in the Java programming language, you'll want to know something about that language in particular and about the modern, networked computing environment for which Java is designed.

As you read this chapter, don't worry if you can't understand everything in detail. (In fact, it would be impossible for you to learn all the details from the brief expositions in this chapter.) Concentrate on learning enough about the big ideas to orient yourself, in preparation for the rest of the book. Most of what is covered in this chapter will be covered in much greater detail later in the book.

1.1 The Fetch and Execute Cycle: Machine Language

A computer is a complex system consisting of many different components. But at the heart—or the brain, if you want—of the computer is a single component that does the actual computing. This is the ***Central Processing Unit***, or CPU. In a modern desktop computer, the CPU is a single "chip" on the order of one square inch in size. The job of the CPU is to execute programs.

A ***program*** is simply a list of unambiguous instructions meant to be followed mechanically by a computer. A computer is built to carry out instructions that are written in a very simple type of language called ***machine language***. Each type of computer has its own machine language, and the computer can directly execute a program only if the program is expressed in that language. (It can execute programs written in other languages if they are first translated into machine language.)

When the CPU executes a program, that program is stored in the computer's ***main memory*** (also called the RAM or random access memory). In addition to the program, memory can also hold data that is being used or processed by the program. Main memory consists of a sequence of ***locations***. These locations are numbered, and the sequence number of a location is called its ***address***. An address provides a way of picking out one particular piece of information from among the millions stored in memory. When the CPU needs to access the program instruction or data in a particular location, it sends the address of that information as a signal to the memory; the memory responds by sending back the data contained in the specified

location. The CPU can also store information in memory by specifying the information to be stored and the address of the location where it is to be stored.

On the level of machine language, the operation of the CPU is fairly straightforward (although it is very complicated in detail). The CPU executes a program that is stored as a sequence of machine language instructions in main memory. It does this by repeatedly reading, or ***fetching***, an instruction from memory and then carrying out, or ***executing***, that instruction. This process—fetch an instruction, execute it, fetch another instruction, execute it, and so on forever—is called the ***fetch-and-execute cycle***. With one exception, which will be covered in the next section, this is all that the CPU ever does.

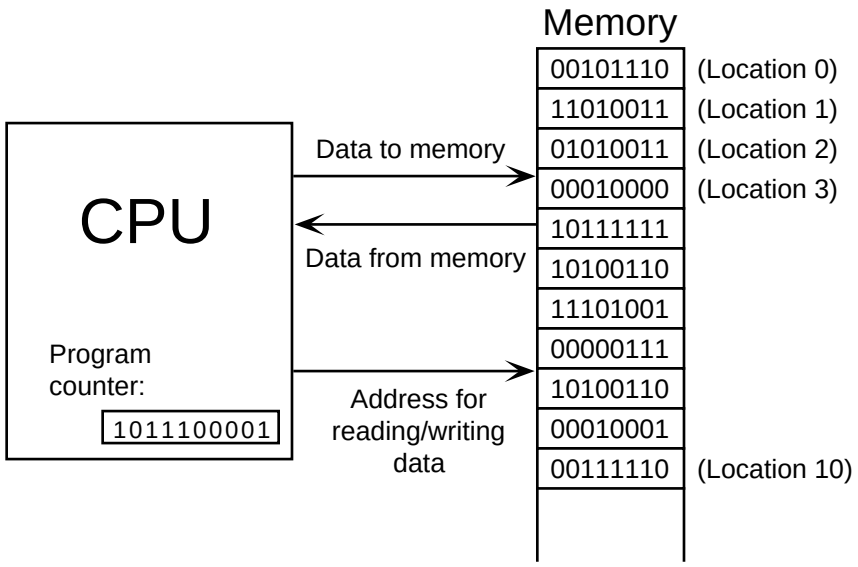
The details of the fetch-and-execute cycle are not terribly important, but there are a few basic things you should know. The CPU contains a few internal ***registers***, which are small memory units capable of holding a single number or machine language instruction. The CPU uses one of these registers—the ***program counter***, or PC—to keep track of where it is in the program it is executing. The PC stores the address of the next instruction that the CPU should execute. At the beginning of each fetch-and-execute cycle, the CPU checks the PC to see which instruction it should fetch. During the course of the fetch-and-execute cycle, the number in the PC is updated to indicate the instruction that is to be executed in the next cycle. (Usually, but not always, this is just the instruction that sequentially follows the current instruction in the program.)

* * *

A computer executes machine language programs mechanically—that is without understanding them or thinking about them—simply because of the way it is physically put together. This is not an easy concept. A computer is a machine built of millions of tiny switches called ***transistors***, which have the property that they can be wired together in such a way that an output from one switch can turn another switch on or off. As a computer computes, these switches turn each other on or off in a pattern determined both by the way they are wired together and by the program that the computer is executing.

Machine language instructions are expressed as binary numbers. A binary number is made up of just two possible digits, zero and one. So, a machine language instruction is just a sequence of zeros and ones. Each particular sequence encodes some particular instruction. The data that the computer manipulates is also encoded as binary numbers. A computer can work directly with binary numbers because switches can readily represent such numbers: Turn the switch on to represent a one; turn it off to represent a zero. Machine language instructions are stored in memory as patterns of switches turned on or off. When a machine language instruction is loaded into the CPU, all that happens is that certain switches are turned on or off in the pattern that encodes that particular instruction. The CPU is built to respond to this pattern by executing the instruction it encodes; it does this simply because of the way all the other switches in the CPU are wired together.

So, you should understand this much about how computers work: Main memory holds machine language programs and data. These are encoded as binary numbers. The CPU fetches machine language instructions from memory one after another and executes them. It does this mechanically, without thinking about or understanding what it does—and therefore the program it executes must be perfect, complete in all details, and unambiguous because the CPU can do nothing but execute it exactly as written. Here is a schematic view of this first-stage understanding of the computer:



1.2 Asynchronous Events: Polling Loops and Interrupts

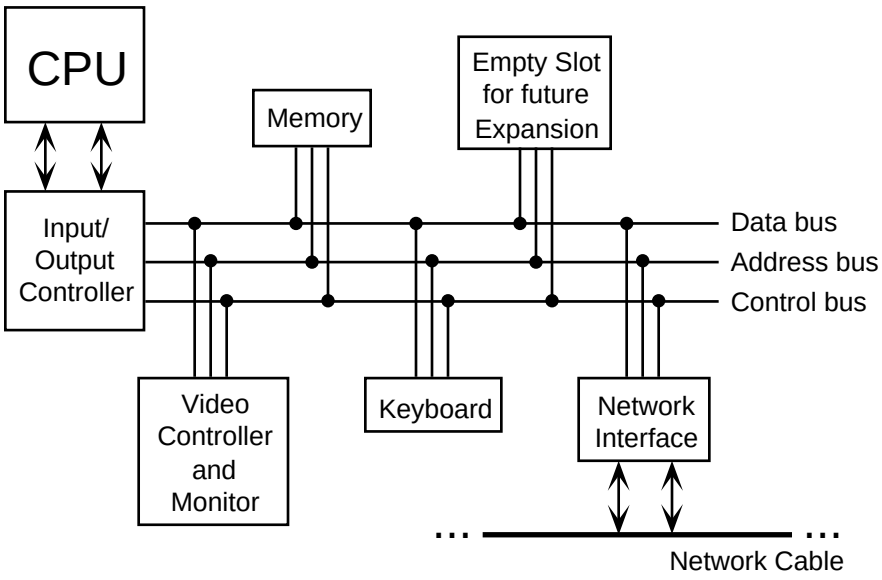
The CPU spends almost all of its time fetching instructions from memory and executing them. However, the CPU and main memory are only two out of many components in a real computer system. A complete system contains other devices such as:

- A ***hard disk*** for storing programs and data files. (Note that main memory holds only a comparatively small amount of information, and holds it only as long as the power is turned on. A hard disk is used for permanent storage of larger amounts of information, but programs have to be loaded from disk into main memory before they can actually be executed.)
- A ***keyboard*** and ***mouse*** for user input.
- A ***monitor*** and ***printer*** which can be used to display the computer's output.
- An ***audio output device*** that allows the computer to play sounds.
- A ***network interface*** that allows the computer to communicate with other computers that are connected to it on a network, either wirelessly or by wire.
- A ***scanner*** that converts images into coded binary numbers that can be stored and manipulated on the computer.

The list of devices is entirely open ended, and computer systems are built so that they can easily be expanded by adding new devices. Somehow the CPU has to communicate with and control all these devices. The CPU can only do this by executing machine language instructions (which is all it can do, period). The way this works is that for each device in a system, there is a ***device driver***, which consists of software that the CPU executes when it has to deal with the device. Installing a new device on a system generally has two steps: plugging the device physically into the computer, and installing the device driver software. Without the device driver, the actual physical device would be useless, since the CPU would not be able to communicate with it.

* * *

A computer system consisting of many devices is typically organized by connecting those devices to one or more ***busses***. A bus is a set of wires that carry various sorts of information between the devices connected to those wires. The wires carry data, addresses, and control signals. An address directs the data to a particular device and perhaps to a particular register or location within that device. Control signals can be used, for example, by one device to alert another that data is available for it on the data bus. A fairly simple computer system might be organized like this:



Now, devices such as keyboard, mouse, and network interface can produce input that needs to be processed by the CPU. How does the CPU know that the data is there? One simple idea, which turns out to be not very satisfactory, is for the CPU to keep checking for incoming data over and over. Whenever it finds data, it processes it. This method is called ***polling***, since the CPU polls the input devices continually to see whether they have any input data to report. Unfortunately, although polling is very simple, it is also very inefficient. The CPU can waste an awful lot of time just waiting for input.

To avoid this inefficiency, ***interrupts*** are often used instead of polling. An interrupt is a signal sent by another device to the CPU. The CPU responds to an interrupt signal by putting aside whatever it is doing in order to respond to the interrupt. Once it has handled the interrupt, it returns to what it was doing before the interrupt occurred. For example, when you press a key on your computer keyboard, a keyboard interrupt is sent to the CPU. The CPU responds to this signal by interrupting what it is doing, reading the key that you pressed, processing it, and then returning to the task it was performing before you pressed the key.

Again, you should understand that this is a purely mechanical process: A device signals an interrupt simply by turning on a wire. The CPU is built so that when that wire is turned on, the CPU saves enough information about what it is currently doing so that it can return to the same state later. This information consists of the contents of important internal registers such as the program counter. Then the CPU jumps to some predetermined memory location and begins executing the instructions stored there. Those instructions make up an ***interrupt handler*** that does the processing necessary to respond to the interrupt. (This interrupt handler is part of the device driver software for the device that signalled the interrupt.) At the end of

the interrupt handler is an instruction that tells the CPU to jump back to what it was doing; it does that by restoring its previously saved state.

Interrupts allow the CPU to deal with ***asynchronous events***. In the regular fetch-and-execute cycle, things happen in a predetermined order; everything that happens is "synchronized" with everything else. Interrupts make it possible for the CPU to deal efficiently with events that happen "asynchronously," that is, at unpredictable times.

As another example of how interrupts are used, consider what happens when the CPU needs to access data that is stored on the hard disk. The CPU can access data directly only if it is in main memory. Data on the disk has to be copied into memory before it can be accessed. Unfortunately, on the scale of speed at which the CPU operates, the disk drive is extremely slow. When the CPU needs data from the disk, it sends a signal to the disk drive telling it to locate the data and get it ready. (This signal is sent synchronously, under the control of a regular program.) Then, instead of just waiting the long and unpredictable amount of time that the disk drive will take to do this, the CPU goes on with some other task. When the disk drive has the data ready, it sends an interrupt signal to the CPU. The interrupt handler can then read the requested data.

* * *

Now, you might have noticed that all this only makes sense if the CPU actually has several tasks to perform. If it has nothing better to do, it might as well spend its time polling for input or waiting for disk drive operations to complete. All modern computers use ***multitasking*** to perform several tasks at once. Some computers can be used by several people at once. Since the CPU is so fast, it can quickly switch its attention from one user to another, devoting a fraction of a second to each user in turn. This application of multitasking is called ***timesharing***. But a modern personal computer with just a single user also uses multitasking. For example, the user might be typing a paper while a clock is continuously displaying the time and a file is being downloaded over the network.

Each of the individual tasks that the CPU is working on is called a ***thread***. (Or a ***process***; there are technical differences between threads and processes, but they are not important here, since it is threads that are used in Java.) Many CPUs can literally execute more than one thread simultaneously—such CPUs contain multiple "cores," each of which can run a thread—but there is always a limit on the number of threads that can be executed at the same time. Since there are often more threads than can be executed simultaneously, the computer has to be able switch its attention from one thread to another, just as a timesharing computer switches its attention from one user to another. In general, a thread that is being executed will continue to run until one of several things happens:

- The thread might voluntarily ***yield*** control, to give other threads a chance to run.
- The thread might have to wait for some asynchronous event to occur. For example, the thread might request some data from the disk drive, or it might wait for the user to press a key. While it is waiting, the thread is said to be ***blocked***, and other threads, if any, have a chance to run. When the event occurs, an interrupt will "wake up" the thread so that it can continue running.
- The thread might use up its allotted slice of time and be suspended to allow other threads to run. Not all computers can "forcibly" suspend a thread in this way; those that can are said to use ***preemptive multitasking***. To do preemptive multitasking, a computer needs a special timer device that generates an interrupt at regular intervals, such as 100 times per second. When a timer interrupt occurs, the CPU has a chance to switch from

one thread to another, whether the thread that is currently running likes it or not. All modern desktop and laptop computers use preemptive multitasking.

Ordinary users, and indeed ordinary programmers, have no need to deal with interrupts and interrupt handlers. They can concentrate on the different tasks or threads that they want the computer to perform; the details of how the computer manages to get all those tasks done are not important to them. In fact, most users, and many programmers, can ignore threads and multitasking altogether. However, threads have become increasingly important as computers have become more powerful and as they have begun to make more use of multitasking and multiprocessing. In fact, the ability to work with threads is fast becoming an essential job skill for programmers. Fortunately, Java has good support for threads, which are built into the Java programming language as a fundamental programming concept. Programming with threads will be covered in Chapter 12.

Just as important in Java and in modern programming in general is the basic concept of asynchronous events. While programmers don't actually deal with interrupts directly, they do often find themselves writing ***event handlers***, which, like interrupt handlers, are called asynchronously when specific events occur. Such "event-driven programming" has a very different feel from the more traditional straight-through, synchronous programming. We will begin with the more traditional type of programming, which is still used for programming individual tasks, but we will return to threads and events later in the text, starting in Chapter 6

* * *

By the way, the software that does all the interrupt handling, handles communication with the user and with hardware devices, and controls which thread is allowed to run is called the ***operating system***. The operating system is the basic, essential software without which a computer would not be able to function. Other programs, such as word processors and World Wide Web browsers, are dependent upon the operating system. Common operating systems include Linux, Windows XP, Windows Vista, and Mac OS.

1.3 The Java Virtual Machine

Machine language consists of very simple instructions that can be executed directly by the CPU of a computer. Almost all programs, though, are written in ***high-level programming languages*** such as Java, Pascal, or C++. A program written in a high-level language cannot be run directly on any computer. First, it has to be translated into machine language. This translation can be done by a program called a ***compiler***. A compiler takes a high-level-language program and translates it into an executable machine-language program. Once the translation is done, the machine-language program can be run any number of times, but of course it can only be run on one type of computer (since each type of computer has its own individual machine language). If the program is to run on another type of computer it has to be re-translated, using a different compiler, into the appropriate machine language.

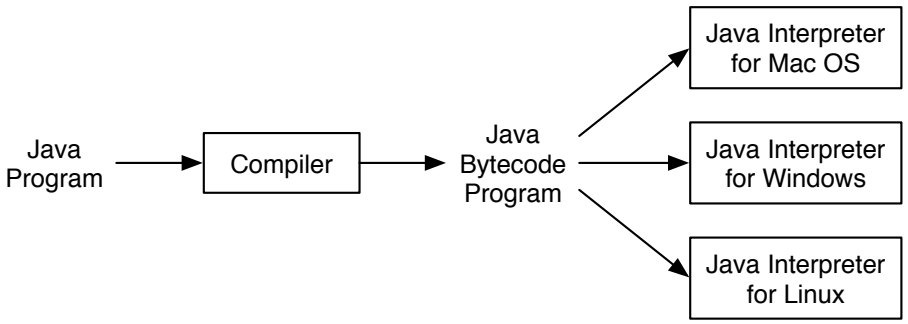
There is an alternative to compiling a high-level language program. Instead of using a compiler, which translates the program all at once, you can use an ***interpreter***, which translates it instruction-by-instruction, as necessary. An interpreter is a program that acts much like a CPU, with a kind of fetch-and-execute cycle. In order to execute a program, the interpreter runs in a loop in which it repeatedly reads one instruction from the program, decides what is necessary to carry out that instruction, and then performs the appropriate machine-language commands to do so.

One use of interpreters is to execute high-level language programs. For example, the programming language Lisp is usually executed by an interpreter rather than a compiler. However, interpreters have another purpose: they can let you use a machine-language program meant for one type of computer on a completely different type of computer. For example, there is a program called "Virtual PC" that runs on Mac OS computers. Virtual PC is an interpreter that executes machine-language programs written for IBM-PC-clone computers. If you run Virtual PC on your Mac OS, you can run any PC program, including programs written for Windows. (Unfortunately, a PC program will run much more slowly than it would on an actual IBM clone. The problem is that Virtual PC executes several Mac OS machine-language instructions for each PC machine-language instruction in the program it is interpreting. Compiled programs are inherently faster than interpreted programs.)

* * *

The designers of Java chose to use a combination of compilation and interpretation. Programs written in Java are compiled into machine language, but it is a machine language for a computer that doesn't really exist. This so-called "virtual" computer is known as the ***Java Virtual Machine***, or JVM. The machine language for the Java Virtual Machine is called ***Java bytecode***. There is no reason why Java bytecode couldn't be used as the machine language of a real computer, rather than a virtual computer. But in fact the use of a virtual machine makes possible one of the main selling points of Java: the fact that it can actually be used on **any** computer. All that the computer needs is an interpreter for Java bytecode. Such an interpreter simulates the JVM in the same way that Virtual PC simulates a PC computer. (The term JVM is also used for the Java bytecode interpreter program that does the simulation, so we say that a computer needs a JVM in order to run Java programs. Technically, it would be more correct to say that the interpreter *implements* the JVM than to say that it *is* a JVM.)

Of course, a different Java bytecode interpreter is needed for each type of computer, but once a computer has a Java bytecode interpreter, it can run any Java bytecode program. And the same Java bytecode program can be run on any computer that has such an interpreter. This is one of the essential features of Java: the same compiled program can be run on many different types of computers.



Why, you might wonder, use the intermediate Java bytecode at all? Why not just distribute the original Java program and let each person compile it into the machine language of whatever computer they want to run it on? There are many reasons. First of all, a compiler has to understand Java, a complex high-level language. The compiler is itself a complex program. A Java bytecode interpreter, on the other hand, is a fairly small, simple program. This makes it easy to write a bytecode interpreter for a new type of computer; once that is done, that computer

can run any compiled Java program. It would be much harder to write a Java compiler for the same computer.

Furthermore, many Java programs are meant to be downloaded over a network. This leads to obvious security concerns: you don't want to download and run a program that will damage your computer or your files. The bytecode interpreter acts as a buffer between you and the program you download. You are really running the interpreter, which runs the downloaded program indirectly. The interpreter can protect you from potentially dangerous actions on the part of that program.

When Java was still a new language, it was criticized for being slow: Since Java bytecode was executed by an interpreter, it seemed that Java bytecode programs could never run as quickly as programs compiled into native machine language (that is, the actual machine language of the computer on which the program is running). However, this problem has been largely overcome by the use of ***just-in-time compilers*** for executing Java bytecode. A just-in-time compiler translates Java bytecode into native machine language. It does this while it is executing the program. Just as for a normal interpreter, the input to a just-in-time compiler is a Java bytecode program, and its task is to execute that program. But as it is executing the program, it also translates parts of it into machine language. The translated parts of the program can then be executed much more quickly than they could be interpreted. Since a given part of a program is often executed many times as the program runs, a just-in-time compiler can significantly speed up the overall execution time.

I should note that there is no necessary connection between Java and Java bytecode. A program written in Java could certainly be compiled into the machine language of a real computer. And programs written in other languages could be compiled into Java bytecode. However, it is the combination of Java and Java bytecode that is platform-independent, secure, and network-compatible while allowing you to program in a modern high-level object-oriented language.

(In the past few years, it has become fairly common to create new programming languages, or versions of old languages, that compile into Java bytecode. The compiled bytecode programs can then be executed by a standard JVM. New languages that have been developed specifically for programming the JVM include Groovy, Clojure, and Processing. Jython and JRuby are versions of older languages, Python and Ruby, that target the JVM. These languages make it possible to enjoy many of the advantages of the JVM while avoiding some of the technicalities of the Java language. In fact, the use of other languages with the JVM has become important enough that several new features have been added to the JVM in Java Version 7 specifically to add better support for some of those languages.)

* * *

I should also note that the really hard part of platform-independence is providing a "Graphical User Interface"—with windows, buttons, etc.—that will work on all the platforms that support Java. You'll see more about this problem in Section 1.6.

1.4 Fundamental Building Blocks of Programs

THERE ARE TWO BASIC ASPECTS of programming: data and instructions. To work with data, you need to understand ***variables*** and ***types***; to work with instructions, you need to understand ***control structures*** and ***subroutines***. You'll spend a large part of the course becoming familiar with these concepts.

A ***variable*** is just a memory location (or several locations treated as a unit) that has been given a name so that it can be easily referred to and used in a program. The programmer only

has to worry about the name; it is the compiler's responsibility to keep track of the memory location. The programmer does need to keep in mind that the name refers to a kind of "box" in memory that can hold data, even if the programmer doesn't have to know where in memory that box is located.

In Java and in many other programming languages, a variable has a ***type*** that indicates what sort of data it can hold. One type of variable might hold integers—whole numbers such as 3, -7, and 0—while another holds floating point numbers—numbers with decimal points such as 3.14, -2.7, or 17.0. (Yes, the computer does make a distinction between the integer 17 and the floating-point number 17.0; they actually look quite different inside the computer.) There could also be types for individual characters ('A', ';', etc.), strings ("Hello", "A string can include many characters", etc.), and less common types such as dates, colors, sounds, or any other kind of data that a program might need to store.

Programming languages always have commands for getting data into and out of variables and for doing computations with data. For example, the following "assignment statement," which might appear in a Java program, tells the computer to take the number stored in the variable named "principal", multiply that number by 0.07, and then store the result in the variable named "interest":

```
interest = principal * 0.07;
```

There are also "input commands" for getting data from the user or from files on the computer's disks and "output commands" for sending data in the other direction.

These basic commands—for moving data from place to place and for performing computations—are the building blocks for all programs. These building blocks are combined into complex programs using control structures and subroutines.

* * *

A program is a sequence of instructions. In the ordinary "flow of control," the computer executes the instructions in the sequence in which they appear, one after the other. However, this is obviously very limited: the computer would soon run out of instructions to execute. ***Control structures*** are special instructions that can change the flow of control. There are two basic types of control structure: ***loops***, which allow a sequence of instructions to be repeated over and over, and ***branches***, which allow the computer to decide between two or more different courses of action by testing conditions that occur as the program is running.

For example, it might be that if the value of the variable "principal" is greater than 10000, then the "interest" should be computed by multiplying the principal by 0.05; if not, then the interest should be computed by multiplying the principal by 0.04. A program needs some way of expressing this type of decision. In Java, it could be expressed using the following "if statement":

```
if (principal > 10000)
    interest = principal * 0.05;
else
    interest = principal * 0.04;
```

(Don't worry about the details for now. Just remember that the computer can test a condition and decide what to do next on the basis of that test.)

Loops are used when the same task has to be performed more than once. For example, if you want to print out a mailing label for each name on a mailing list, you might say, "Get the first name and address and print the label; get the second name and address and print the label; get the third name and address and print the label. . . " But this quickly becomes

ridiculous—and might not work at all if you don't know in advance how many names there are. What you would like to say is something like "While there are more names to process, get the next name and address, and print the label." A loop can be used in a program to express such repetition.

* * *

Large programs are so complex that it would be almost impossible to write them if there were not some way to break them up into manageable "chunks." Subroutines provide one way to do this. A ***subroutine*** consists of the instructions for performing some task, grouped together as a unit and given a name. That name can then be used as a substitute for the whole set of instructions. For example, suppose that one of the tasks that your program needs to perform is to draw a house on the screen. You can take the necessary instructions, make them into a subroutine, and give that subroutine some appropriate name—say, "drawHouse()". Then anyplace in your program where you need to draw a house, you can do so with the single command:

```
drawHouse();
```

This will have the same effect as repeating all the house-drawing instructions in each place.

The advantage here is not just that you save typing. Organizing your program into subroutines also helps you organize your thinking and your program design effort. While writing the house-drawing subroutine, you can concentrate on the problem of drawing a house without worrying for the moment about the rest of the program. And once the subroutine is written, you can forget about the details of drawing houses—that problem is solved, since you have a subroutine to do it for you. A subroutine becomes just like a built-in part of the language which you can use without thinking about the details of what goes on "inside" the subroutine.

* * *

Variables, types, loops, branches, and subroutines are the basis of what might be called "traditional programming." However, as programs become larger, additional structure is needed to help deal with their complexity. One of the most effective tools that has been found is object-oriented programming, which is discussed in the next section.

1.5 Objects and Object-oriented Programming

PROGRAMS MUST BE DESIGNED. No one can just sit down at the computer and compose a program of any complexity. The discipline called ***software engineering*** is concerned with the construction of correct, working, well-written programs. The software engineer tries to use accepted and proven methods for analyzing the problem to be solved and for designing a program to solve that problem.

During the 1970s and into the 80s, the primary software engineering methodology was ***structured programming***. The structured programming approach to program design was based on the following advice: To solve a large problem, break the problem into several pieces and work on each piece separately; to solve each piece, treat it as a new problem which can itself be broken down into smaller problems; eventually, you will work your way down to problems that can be solved directly, without further decomposition. This approach is called ***top-down programming***.

There is nothing wrong with top-down programming. It is a valuable and often-used approach to problem-solving. However, it is incomplete. For one thing, it deals almost entirely with producing the **instructions** necessary to solve a problem. But as time went on, people

realized that the design of the **data structures** for a program was at least as important as the design of subroutines and control structures. Top-down programming doesn't give adequate consideration to the data that the program manipulates.

Another problem with strict top-down programming is that it makes it difficult to reuse work done for other projects. By starting with a particular problem and subdividing it into convenient pieces, top-down programming tends to produce a design that is unique to that problem. It is unlikely that you will be able to take a large chunk of programming from another program and fit it into your project, at least not without extensive modification. Producing high-quality programs is difficult and expensive, so programmers and the people who employ them are always eager to reuse past work.

* * *

So, in practice, top-down design is often combined with ***bottom-up design***. In bottom-up design, the approach is to start "at the bottom," with problems that you already know how to solve (and for which you might already have a reusable software component at hand). From there, you can work upwards towards a solution to the overall problem.

The reusable components should be as "modular" as possible. A ***module*** is a component of a larger system that interacts with the rest of the system in a simple, well-defined, straightforward manner. The idea is that a module can be "plugged into" a system. The details of what goes on inside the module are not important to the system as a whole, as long as the module fulfills its assigned role correctly. This is called ***information hiding***, and it is one of the most important principles of software engineering.

One common format for software modules is to contain some data, along with some subroutines for manipulating that data. For example, a mailing-list module might contain a list of names and addresses along with a subroutine for adding a new name, a subroutine for printing mailing labels, and so forth. In such modules, the data itself is often hidden inside the module; a program that uses the module can then manipulate the data only indirectly, by calling the subroutines provided by the module. This protects the data, since it can only be manipulated in known, well-defined ways. And it makes it easier for programs to use the module, since they don't have to worry about the details of how the data is represented. Information about the representation of the data is hidden.

Modules that could support this kind of information-hiding became common in programming languages in the early 1980s. Since then, a more advanced form of the same idea has more or less taken over software engineering. This latest approach is called ***object-oriented programming***, often abbreviated as OOP.

The central concept of object-oriented programming is the ***object***, which is a kind of module containing data and subroutines. The point-of-view in OOP is that an object is a kind of self-sufficient entity that has an internal ***state*** (the data it contains) and that can respond to ***messages*** (calls to its subroutines). A mailing list object, for example, has a state consisting of a list of names and addresses. If you send it a message telling it to add a name, it will respond by modifying its state to reflect the change. If you send it a message telling it to print itself, it will respond by printing out its list of names and addresses.

The OOP approach to software engineering is to start by identifying the objects involved in a problem and the messages that those objects should respond to. The program that results is a collection of objects, each with its own data and its own set of responsibilities. The objects interact by sending messages to each other. There is not much "top-down" in the large-scale design of such a program, and people used to more traditional programs can have a hard time getting used to OOP. However, people who use OOP would claim that object-oriented programs

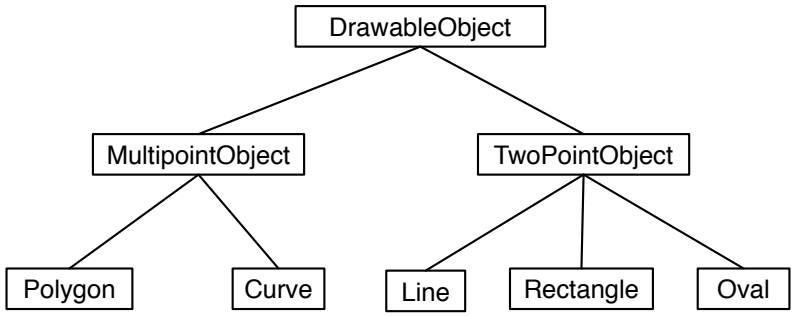
tend to be better models of the way the world itself works, and that they are therefore easier to write, easier to understand, and more likely to be correct.

* * *

You should think of objects as "knowing" how to respond to certain messages. Different objects might respond to the same message in different ways. For example, a "print" message would produce very different results, depending on the object it is sent to. This property of objects—that different objects can respond to the same message in different ways—is called ***polymorphism***.

It is common for objects to bear a kind of "family resemblance" to one another. Objects that contain the same type of data and that respond to the same messages in the same way belong to the same ***class***. (In actual programming, the class is primary; that is, a class is created and then one or more objects are created using that class as a template.) But objects can be similar without being in exactly the same class.

For example, consider a drawing program that lets the user draw lines, rectangles, ovals, polygons, and curves on the screen. In the program, each visible object on the screen could be represented by a software object in the program. There would be five classes of objects in the program, one for each type of visible object that can be drawn. All the lines would belong to one class, all the rectangles to another class, and so on. These classes are obviously related; all of them represent "drawable objects." They would, for example, all presumably be able to respond to a "draw yourself" message. Another level of grouping, based on the data needed to represent each type of object, is less obvious, but would be very useful in a program: We can group polygons and curves together as "multipoint objects," while lines, rectangles, and ovals are "two-point objects." (A line is determined by its endpoints, a rectangle by two of its corners, and an oval by two corners of the rectangle that contains it.) We could diagram these relationships as follows:



DrawableObject, MultipointObject, and TwoPointObject would be classes in the program. MultipointObject and TwoPointObject would be ***subclasses*** of DrawableObject. The class Line would be a subclass of TwoPointObject and (indirectly) of DrawableObject. A subclass of a class is said to ***inherit*** the properties of that class. The subclass can add to its inheritance and it can even "override" part of that inheritance (by defining a different response to some method). Nevertheless, lines, rectangles, and so on **are** drawable objects, and the class DrawableObject expresses this relationship.

Inheritance is a powerful means for organizing a program. It is also related to the problem of reusing software components. A class is the ultimate reusable component. Not only can it be reused directly if it fits exactly into a program you are trying to write, but if it just almost

fits, you can still reuse it by defining a subclass and making only the small changes necessary to adapt it exactly to your needs.

So, OOP is meant to be both a superior program-development tool and a partial solution to the software reuse problem. Objects, classes, and object-oriented programming will be important themes throughout the rest of this text. You will start using objects that are built into the Java language in the next chapter, and in Chapter 5 you will begin creating your own classes and objects.

1.6 The Modern User Interface

When computers were first introduced, ordinary people—including most programmers—couldn't get near them. They were locked up in rooms with white-coated attendants who would take your programs and data, feed them to the computer, and return the computer's response some time later. When timesharing—where the computer switches its attention rapidly from one person to another—was invented in the 1960s, it became possible for several people to interact directly with the computer at the same time. On a timesharing system, users sit at "terminals" where they type commands to the computer, and the computer types back its response. Early personal computers also used typed commands and responses, except that there was only one person involved at a time. This type of interaction between a user and a computer is called a ***command-line interface***.

Today, of course, most people interact with computers in a completely different way. They use a ***Graphical User Interface***, or GUI. The computer draws interface ***components*** on the screen. The components include things like windows, scroll bars, menus, buttons, and icons. Usually, a ***mouse*** is used to manipulate such components. Assuming that you have not just been teleported in from the 1970s, you are no doubt already familiar with the basics of graphical user interfaces!

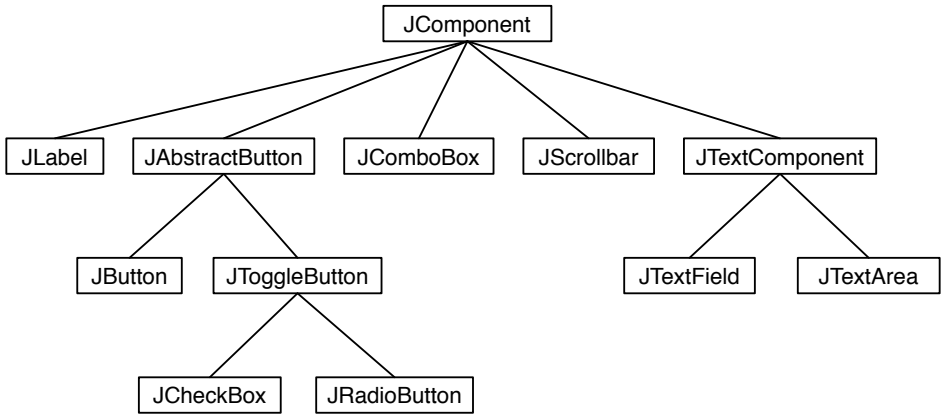
A lot of GUI interface components have become fairly standard. That is, they have similar appearance and behavior on many different computer platforms including Mac OS, Windows, and Linux. Java programs, which are supposed to run on many different platforms without modification to the program, can use all the standard GUI components. They might vary a little in appearance from platform to platform, but their functionality should be identical on any computer on which the program runs.

Shown below is an image of a very simple Java program—actually an "***applet***", since it is meant to appear on a Web page—that shows a few standard GUI interface components. There are four components that the user can interact with: a button, a checkbox, a text field, and a pop-up menu. These components are labeled. There are a few other components in the applet. The labels themselves are components (even though you can't interact with them). The right half of the applet is a text area component, which can display multiple lines of text. And a scrollbar component appears alongside the text area when the number of lines of text becomes larger than will fit in the text area. And in fact, in Java terminology, the whole applet is itself considered to be a "component."

Now, Java actually has two complete sets of GUI components. One of these, the AWT or ***Abstract Windowing Toolkit***, was available in the original version of Java. The other, which is known as ***Swing***, is included in Java version 1.2 or later, and is used in preference to the AWT in most modern Java programs. The applet that is shown above uses components that are part of Swing. If Java is not installed in your Web browser or if your browser uses a very old version of Java, you might get an error when the browser tries to load the applet. Remember that most of the applets in this textbook require Java 5.0 (or higher).

When a user interacts with the GUI components in this applet, an "event" is generated. For example, clicking a push button generates an event, and pressing return while typing in a text field generates an event. Each time an event is generated, a message is sent to the applet telling it that the event has occurred, and the applet responds according to its program. In fact, the program consists mainly of "event handlers" that tell the applet how to respond to various types of events. In this example, the applet has been programmed to respond to each event by displaying a message in the text area. In a more realistic example, the event handlers would have more to do.

The use of the term "message" here is deliberate. Messages, as you saw in the previous section, are sent to objects. In fact, Java GUI components are implemented as objects. Java includes many predefined classes that represent various types of GUI components. Some of these classes are subclasses of others. Here is a diagram showing some of Swing's GUI classes and their relationships:



Don't worry about the details for now, but try to get some feel about how object-oriented programming and inheritance are used here. Note that all the GUI classes are subclasses, directly or indirectly, of a class called *JComponent*, which represents general properties that are shared by all Swing components. Two of the direct subclasses of *JComponent* themselves have subclasses. The classes *JTextArea* and *JTextField*, which have certain behaviors in common, are grouped together as subclasses of *JTextComponent*. Similarly *JButton* and *JToggleButton*

are subclasses of *JAbstractButton*, which represents properties common to both buttons and checkboxes. (*JComboBox*, by the way, is the Swing class that represents pop-up menus.)

Just from this brief discussion, perhaps you can see how GUI programming can make effective use of object-oriented design. In fact, GUI's, with their "visible objects," are probably a major factor contributing to the popularity of OOP.

Programming with GUI components and events is one of the most interesting aspects of Java. However, we will spend several chapters on the basics before returning to this topic in Chapter 6.

1.7 The Internet and Beyond

COMPUTERS CAN BE CONNECTED together on ***networks***. A computer on a network can communicate with other computers on the same network by exchanging data and files or by sending and receiving messages. Computers on a network can even work together on a large computation.

Today, millions of computers throughout the world are connected to a single huge network called the ***Internet***. New computers are being connected to the Internet every day, both by wireless communication and by physical connection using technologies such as DSL, cable modems, or Ethernet.

There are elaborate ***protocols*** for communication over the Internet. A protocol is simply a detailed specification of how communication is to proceed. For two computers to communicate at all, they must both be using the same protocols. The most basic protocols on the Internet are the ***Internet Protocol*** (IP), which specifies how data is to be physically transmitted from one computer to another, and the ***Transmission Control Protocol*** (TCP), which ensures that data sent using IP is received in its entirety and without error. These two protocols, which are referred to collectively as TCP/IP, provide a foundation for communication. Other protocols use TCP/IP to send specific types of information such as web pages, electronic mail, and data files.

All communication over the Internet is in the form of ***packets***. A packet consists of some data being sent from one computer to another, along with addressing information that indicates where on the Internet that data is supposed to go. Think of a packet as an envelope with an address on the outside and a message on the inside. (The message is the data.) The packet also includes a "return address," that is, the address of the sender. A packet can hold only a limited amount of data; longer messages must be divided among several packets, which are then sent individually over the net and reassembled at their destination.

Every computer on the Internet has an ***IP address***, a number that identifies it uniquely among all the computers on the net. The IP address is used for addressing packets. A computer can only send data to another computer on the Internet if it knows that computer's IP address. Since people prefer to use names rather than numbers, most computers are also identified by names, called ***domain names***. For example, the main computer of the Mathematics Department at Hobart and William Smith Colleges has the domain name math.hws.edu. (Domain names are just for convenience; your computer still needs to know IP addresses before it can communicate. There are computers on the Internet whose job it is to translate domain names to IP addresses. When you use a domain name, your computer sends a message to a domain name server to find out the corresponding IP address. Then, your computer uses the IP address, rather than the domain name, to communicate with the other computer.)

The Internet provides a number of services to the computers connected to it (and, of course,

to the users of those computers). These services use TCP/IP to send various types of data over the net. Among the most popular services are instant messaging, file sharing, electronic mail, and the World-Wide Web. Each service has its own protocols, which are used to control transmission of data over the network. Each service also has some sort of user interface, which allows the user to view, send, and receive data through the service.

For example, the email service uses a protocol known as ***SMTP*** (Simple Mail Transfer Protocol) to transfer email messages from one computer to another. Other protocols, such as POP and IMAP, are used to fetch messages from an email account so that the recipient can read them. A person who uses email, however, doesn't need to understand or even know about these protocols. Instead, they are used behind the scenes by computer programs to send and receive email messages. These programs provide the user with an easy-to-use user interface to the underlying network protocols.

The World-Wide Web is perhaps the most exciting of network services. The World-Wide Web allows you to request ***pages*** of information that are stored on computers all over the Internet. A Web page can contain ***links*** to other pages on the same computer from which it was obtained or to other computers anywhere in the world. A computer that stores such pages of information is called a ***web server***. The user interface to the Web is the type of program known as a ***web browser***. Common web browsers include Internet Explorer and Firefox. You use a Web browser to request a page of information. The browser sends a request for that page to the computer on which the page is stored, and when a response is received from that computer, the web browser displays it to you in a neatly formatted form. A web browser is just a user interface to the Web. Behind the scenes, the web browser uses a protocol called ***HTTP*** (HyperText Transfer Protocol) to send each page request and to receive the response from the web server.

* * *

Now just what, you might be thinking, does all this have to do with Java? In fact, Java is intimately associated with the Internet and the World-Wide Web. As you have seen in the previous section, special Java programs called applets are meant to be transmitted over the Internet and displayed on Web pages. A Web server transmits a Java applet just as it would transmit any other type of information. A Web browser that understands Java—that is, that includes an interpreter for the Java Virtual Machine—can then run the applet right on the Web page. Since applets are programs, they can do almost anything, including complex interaction with the user. With Java, a Web page becomes more than just a passive display of information. It becomes anything that programmers can imagine and implement.

But applets are only one aspect of Java's relationship with the Internet, and not the major one. In fact, as both Java and the Internet have matured, applets have become much less important. At the same time, however, Java has increasingly been used to write complex, stand-alone applications that do not depend on a Web browser. Many of these programs are network-related. For example many of the largest and most complex web sites use web server software that is written in Java. Java includes excellent support for network protocols, and its platform independence makes it possible to write network programs that work on many different types of computer. You will learn about Java's network support in Chapter 11.

Its association with the Internet is not Java's only advantage. But many good programming languages have been invented only to be soon forgotten. Java has had the good luck to ride on the coattails of the Internet's immense and increasing popularity.

* * *

As Java has matured, its applications have reached far beyond the Net. The standard version

of Java already comes with support for many technologies, such as cryptography and data compression. Free extensions are available to support many other technologies such as advanced sound processing and three-dimensional graphics. Complex, high-performance systems can be developed in Java. For example, Hadoop, a system for large scale data processing, is written in Java. Hadoop is used by Yahoo, Facebook, and other Web sites to process the huge amounts of data generated by their users.

Furthermore, Java is not restricted to use on traditional computers. Java can be used to write programs for many smartphones (though not for the iPhone). It is the primary development language for Blackberries and Android-based phones such as the Verizon Droid. Mobile devices such as smartphones use a version of Java called Java ME ("Mobile Edition"). It's the same basic language as the standard edition, but the set of classes that is included as a standard part of the language is different. Java ME is also the programming language for the Amazon Kindle eBook reader and for interactive features on Blu-Ray video disks.

At this time, Java certainly ranks as one of the most widely used programming languages. It is a good choice for almost any programming project that is meant to run on more than one type of computing device, and is a reasonable choice even for many programs that will run on only one device. It is probably the most widely taught language at Colleges and Universities. It is similar enough to other popular languages, such as C, C++, and C#, that knowing it will give you a good start on learning those languages as well. Overall, learning Java is a great starting point on the road to becoming an expert programmer. I hope you enjoy the journey!

Quiz on Chapter 1

1. One of the components of a computer is its *CPU.* What is a CPU and what role does it play in a computer?

2. Explain what is meant by an "asynchronous event." Give some examples.

3. What is the difference between a "compiler" and an "interpreter"?

4. Explain the difference between *high-level languages* and *machine language.*

5. If you have the source code for a Java program, and you want to run that program, you will need both a *compiler* and an *interpreter.* What does the Java compiler do, and what does the Java interpreter do?

6. What is a *subroutine?*

7. Java is an object-oriented programming language. What is an *object*?

8. What is a *variable?* (There are four different ideas associated with variables in Java. Try to mention all four aspects in your answer. Hint: One of the aspects is the variable's name.)

9. Java is a "platform-independent language." What does this mean?

10. What is the "Internet"? Give some examples of how it is used. (What kind of services does it provide?)

Chapter 2

Programming in the Small I: Names and Things

On a basic level (the level of machine language), a computer can perform only very simple operations. A computer performs complex tasks by stringing together large numbers of such operations. Such tasks must be "scripted" in complete and perfect detail by programs. Creating complex programs will never be really easy, but the difficulty can be handled to some extent by giving the program a clear overall ***structure***. The design of the overall structure of a program is what I call "programming in the large."

Programming in the small, which is sometimes called ***coding***, would then refer to filling in the details of that design. The details are the explicit, step-by-step instructions for performing fairly small-scale tasks. When you do coding, you are working fairly "close to the machine," with some of the same concepts that you might use in machine language: memory locations, arithmetic operations, loops and branches. In a high-level language such as Java, you get to work with these concepts on a level several steps above machine language. However, you still have to worry about getting all the details exactly right.

This chapter and the next examine the facilities for programming in the small in the Java programming language. Don't be misled by the term "programming in the small" into thinking that this material is easy or unimportant. This material is an essential foundation for all types of programming. If you don't understand it, you can't write programs, no matter how good you get at designing their large-scale structure.

The last section of this chapter discusses ***programming environments***. That section contains information about how to compile and run Java programs, and you might want to take a look at it before trying to write and use your own programs.

2.1 The Basic Java Application

A program is a sequence of instructions that a computer can execute to perform some task. A simple enough idea, but for the computer to make any use of the instructions, they must be written in a form that the computer can use. This means that programs have to be written in ***programming languages***. Programming languages differ from ordinary human languages in being completely unambiguous and very strict about what is and is not allowed in a program. The rules that determine what is allowed are called the ***syntax*** of the language. Syntax rules specify the basic vocabulary of the language and how programs can be constructed using things like loops, branches, and subroutines. A syntactically correct program is one that

can be successfully compiled or interpreted; programs that have syntax errors will be rejected (hopefully with a useful error message that will help you fix the problem).

So, to be a successful programmer, you have to develop a detailed knowledge of the syntax of the programming language that you are using. However, syntax is only part of the story. It's not enough to write a program that will run—you want a program that will run and produce the correct result! That is, the **meaning** of the program has to be right. The meaning of a program is referred to as its ***semantics***. A semantically correct program is one that does what you want it to.

Furthermore, a program can be syntactically and semantically correct but still be a pretty bad program. Using the language correctly is not the same as using it **well**. For example, a good program has "style." It is written in a way that will make it easy for people to read and to understand. It follows conventions that will be familiar to other programmers. And it has an overall design that will make sense to human readers. The computer is completely oblivious to such things, but to a human reader, they are paramount. These aspects of programming are sometimes referred to as ***pragmatics.***

When I introduce a new language feature, I will explain the syntax, the semantics, and some of the pragmatics of that feature. You should memorize the syntax; that's the easy part. Then you should get a feeling for the semantics by following the examples given, making sure that you understand how they work, and maybe writing short programs of your own to test your understanding. And you should try to appreciate and absorb the pragmatics—this means learning how to use the language feature *well*, with style that will earn you the admiration of other programmers.

Of course, even when you've become familiar with all the individual features of the language, that doesn't make you a programmer. You still have to learn how to construct complex programs to solve particular problems. For that, you'll need both experience and taste. You'll find hints about software development throughout this textbook.

* * *

We begin our exploration of Java with the problem that has become traditional for such beginnings: to write a program that displays the message "Hello World!". This might seem like a trivial problem, but getting a computer to do this is really a big first step in learning a new programming language (especially if it's your first programming language). It means that you understand the basic process of:

1. getting the program text into the computer,

2. compiling the program, and

3. running the compiled program.

The first time through, each of these steps will probably take you a few tries to get right. I won't go into the details here of how you do each of these steps; it depends on the particular computer and Java programming environment that you are using. See Section 2.6 for information about creating and running Java programs in specific programming environments. But in general, you will type the program using some sort of text editor and save the program in a file. Then, you will use some command to try to compile the file. You'll either get a message that the program contains syntax errors, or you'll get a compiled version of the program. In the case of Java, the program is compiled into Java bytecode, not into machine language. Finally, you can run the compiled program by giving some appropriate command. For Java, you will actually use an interpreter to execute the Java bytecode. Your programming environment might automate

some of the steps for you—for example, the compilation step is often done automatically—but you can be sure that the same three steps are being done in the background.

Here is a Java program to display the message "Hello World!". Don't expect to understand what's going on here just yet; some of it you won't really understand until a few chapters from now:

```
// A program to display the message
// "Hello World!" on standard output

public class HelloWorld {

   public static void main(String[] args) {
      System.out.println("Hello World!");
   }

}   // end of class HelloWorld
```

The command that actually displays the message is:

```
System.out.println("Hello World!");
```

This command is an example of a ***subroutine call statement***. It uses a "built-in subroutine" named `System.out.println` to do the actual work. Recall that a subroutine consists of the instructions for performing some task, chunked together and given a name. That name can be used to "call" the subroutine whenever that task needs to be performed. A ***built-in subroutine*** is one that is already defined as part of the language and therefore automatically available for use in any program.

When you run this program, the message "Hello World!" (without the quotes) will be displayed on standard output. Unfortunately, I can't say exactly what that means! Java is meant to run on many different platforms, and standard output will mean different things on different platforms. However, you can expect the message to show up in some convenient place. (If you use a command-line interface, like that in Oracle's Java Development Kit, you type in a command to tell the computer to run the program. The computer will type the output from the program, Hello World!, on the next line. In an integrated development environment such as Eclipse, the output might appear somewhere in one of the environment's windows.)

You must be curious about all the other stuff in the above program. Part of it consists of ***comments***. Comments in a program are entirely ignored by the computer; they are there for human readers only. This doesn't mean that they are unimportant. Programs are meant to be read by people as well as by computers, and without comments, a program can be very difficult to understand. Java has two types of comments. The first type, used in the above program, begins with `//` and extends to the end of a line. The computer ignores the `//` and everything that follows it on the same line. Java has another style of comment that can extend over many lines. That type of comment begins with `/*` and ends with `*/`.

Everything else in the program is required by the rules of Java syntax. All programming in Java is done inside "classes." The first line in the above program (not counting the comments) says that this is a class named *HelloWorld*. "HelloWorld," the name of the class, also serves as the name of the program. Not every class is a program. In order to define a program, a class must include a subroutine named `main`, with a definition that takes the form:

```
public static void main(String[] args) {
      ⟨statements⟩
}
```

When you tell the Java interpreter to run the program, the interpreter calls this `main()` subroutine, and the statements that it contains are executed. These statements make up the script that tells the computer exactly what to do when the program is executed. The `main()` routine can call subroutines that are defined in the same class or even in other classes, but it is the `main()` routine that determines how and in what order the other subroutines are used.

The word "public" in the first line of `main()` means that this routine can be called from outside the program. This is essential because the `main()` routine is called by the Java interpreter, which is something external to the program itself. The remainder of the first line of the routine is harder to explain at the moment; for now, just think of it as part of the required syntax. The definition of the subroutine—that is, the instructions that say what it does—consists of the sequence of "statements" enclosed between braces, `{` and `}`. Here, I've used ⟨*statements*⟩ as a placeholder for the actual statements that make up the program. Throughout this textbook, I will always use a similar format: anything that you see in ⟨*this style of text*⟩ (italic in angle brackets) is a placeholder that describes something you need to type when you write an actual program.

As noted above, a subroutine can't exist by itself. It has to be part of a "class". A program is defined by a public class that takes the form:

```
public class ⟨program-name⟩ {

    ⟨optional-variable-declarations-and-subroutines⟩

    public static void main(String[] args) {
       ⟨statements⟩
    }

    ⟨optional-variable-declarations-and-subroutines⟩

}
```

The name on the first line is the name of the program, as well as the name of the class. (Remember, again, that ⟨*program-name*⟩ is a placeholder for the actual name!) If the name of the class is HelloWorld, then the class must be saved in a file called `HelloWorld.java`. When this file is compiled, another file named `HelloWorld.class` will be produced. This class file, `HelloWorld.class`, contains the translation of the program into Java bytecode, which can be executed by a Java interpreter. `HelloWorld.java` is called the ***source code*** for the program. To execute the program, you only need the compiled `class` file, not the source code.

The layout of the program on the page, such as the use of blank lines and indentation, is not part of the syntax or semantics of the language. The computer doesn't care about layout—you could run the entire program together on one line as far as it is concerned. However, layout is important to human readers, and there are certain style guidelines for layout that are followed by most programmers. These style guidelines are part of the pragmatics of the Java programming language.

Also note that according to the above syntax specification, a program can contain other subroutines besides `main()`, as well as things called "variable declarations." You'll learn more about these later, but not until Chapter 4.

2.2 Variables and the Primitive Types

Names are fundamental to programming. In programs, names are used to refer to many different sorts of things. In order to use those things, a programmer must understand the rules

for giving names to things and the rules for using the names to work with those things. That is, the programmer must understand the syntax and the semantics of names.

According to the syntax rules of Java, a name is a sequence of one or more characters. It must begin with a letter or underscore and must consist entirely of letters, digits, and underscores. ("Underscore" refers to the character '_'.) For example, here are some legal names:

```
N   n   rate  x15   quite_a_long_name   HelloWorld
```

No spaces are allowed in identifiers; `HelloWorld` is a legal identifier, but "Hello World" is not. Upper case and lower case letters are considered to be different, so that `HelloWorld`, `helloworld`, `HELLOWORLD`, and `hElloWorLD` are all distinct names. Certain names are reserved for special uses in Java, and cannot be used by the programmer for other purposes. These ***reserved words*** include: `class`, `public`, `static`, `if`, `else`, `while`, and several dozen other words.

Java is actually pretty liberal about what counts as a letter or a digit. Java uses the ***Unicode*** character set, which includes thousands of characters from many different languages and different alphabets, and many of these characters count as letters or digits. However, I will be sticking to what can be typed on a regular English keyboard.

The pragmatics of naming includes style guidelines about how to choose names for things. For example, it is customary for names of classes to begin with upper case letters, while names of variables and of subroutines begin with lower case letters; you can avoid a lot of confusion by following the same convention in your own programs. Most Java programmers do not use underscores in names, although some do use them at the beginning of the names of certain kinds of variables. When a name is made up of several words, such as `HelloWorld` or `interestRate`, it is customary to capitalize each word, except possibly the first; this is sometimes referred to as ***camel case***, since the upper case letters in the middle of a name are supposed to look something like the humps on a camel's back.

Finally, I'll note that things are often referred to by ***compound names*** which consist of several ordinary names separated by periods. (Compound names are also called ***qualified names***.) You've already seen an example: `System.out.println`. The idea here is that things in Java can contain other things. A compound name is a kind of path to an item through one or more levels of containment. The name `System.out.println` indicates that something called "System" contains something called "out" which in turn contains something called "println". Non-compound names are called ***simple identifiers***. I'll use the term ***identifier*** to refer to any name—simple or compound—that can be used to refer to something in Java. (Note that the reserved words are **not** identifiers, since they can't be used as names for things.)

2.2.1 Variables

Programs manipulate data that are stored in memory. In machine language, data can only be referred to by giving the numerical address of the location in memory where it is stored. In a high-level language such as Java, names are used instead of numbers to refer to data. It is the job of the computer to keep track of where in memory the data is actually stored; the programmer only has to remember the name. A name used in this way—to refer to data stored in memory—is called a ***variable***.

Variables are actually rather subtle. Properly speaking, a variable is not a name for the data itself but for a location in memory that can hold data. You should think of a variable as a container or box where you can store data that you will need to use later. The variable refers directly to the box and only indirectly to the data in the box. Since the data in the box can

change, a variable can refer to different data values at different times during the execution of the program, but it always refers to the same box. Confusion can arise, especially for beginning programmers, because when a variable is used in a program in certain ways, it refers to the container, but when it is used in other ways, it refers to the data in the container. You'll see examples of both cases below.

(In this way, a variable is something like the title, "The President of the United States." This title can refer to different people at different times, but it always refers to the same office. If I say "the President is playing basketball," I mean that Barack Obama is playing basketball. But if I say "Sarah Palin wants to be President" I mean that she wants to fill the office, not that she wants to be Barack Obama.)

In Java, the **only** way to get data into a variable—that is, into the box that the variable names—is with an ***assignment statement***. An assignment statement takes the form:

⟨*variable*⟩ = ⟨*expression*⟩;

where ⟨*expression*⟩ represents anything that refers to or computes a data value. When the computer comes to an assignment statement in the course of executing a program, it evaluates the expression and puts the resulting data value into the variable. For example, consider the simple assignment statement

```
rate = 0.07;
```

The ⟨*variable*⟩ in this assignment statement is `rate`, and the ⟨*expression*⟩ is the number 0.07. The computer executes this assignment statement by putting the number 0.07 in the variable `rate`, replacing whatever was there before. Now, consider the following more complicated assignment statement, which might come later in the same program:

```
interest = rate * principal;
```

Here, the value of the expression "`rate * principal`" is being assigned to the variable `interest`. In the expression, the `*` is a "multiplication operator" that tells the computer to multiply `rate` times `principal`. The names `rate` and `principal` are themselves variables, and it is really the **values** stored in those variables that are to be multiplied. We see that when a variable is used in an expression, it is the value stored in the variable that matters; in this case, the variable seems to refer to the data in the box, rather than to the box itself. When the computer executes this assignment statement, it takes the **value** of `rate`, multiplies it by the **value** of `principal`, and stores the answer in the **box** referred to by `interest`. When a variable is used on the left-hand side of an assignment statement, it refers to the box that is named by the variable.

(Note, by the way, that an assignment statement is a command that is executed by the computer at a certain time. It is not a statement of fact. For example, suppose a program includes the statement "`rate = 0.07;`". If the statement "`interest = rate * principal;`" is executed later in the program, can we say that the `principal` is multiplied by 0.07? No! The value of `rate` might have been changed in the meantime by another statement. The meaning of an assignment statement is completely different from the meaning of an equation in mathematics, even though both use the symbol "=".)

2.2.2 Types and Literals

A variable in Java is designed to hold only one particular type of data; it can legally hold that type of data and no other. The compiler will consider it to be a syntax error if you try to violate this rule. We say that Java is a ***strongly typed*** language because it enforces this rule.

There are eight so-called ***primitive types*** built into Java. The primitive types are named **byte**, **short**, **int**, **long**, **float**, **double**, **char**, and **boolean**. The first four types hold integers (whole numbers such as 17, -38477, and 0). The four integer types are distinguished by the ranges of integers they can hold. The **float** and **double** types hold real numbers (such as 3.6 and -145.99). Again, the two real types are distinguished by their range and accuracy. A variable of type **char** holds a single character from the Unicode character set. And a variable of type **boolean** holds one of the two logical values `true` or `false`.

Any data value stored in the computer's memory must be represented as a binary number, that is as a string of zeros and ones. A single zero or one is called a ***bit***. A string of eight bits is called a ***byte***. Memory is usually measured in terms of bytes. Not surprisingly, the **byte** data type refers to a single byte of memory. A variable of type **byte** holds a string of eight bits, which can represent any of the integers between -128 and 127, inclusive. (There are 256 integers in that range; eight bits can represent 256—two raised to the power eight—different values.) As for the other integer types,

- **short** corresponds to two bytes (16 bits). Variables of type **short** have values in the range -32768 to 32767.
- **int** corresponds to four bytes (32 bits). Variables of type **int** have values in the range -2147483648 to 2147483647.
- **long** corresponds to eight bytes (64 bits). Variables of type **long** have values in the range -9223372036854775808 to 9223372036854775807.

You don't have to remember these numbers, but they do give you some idea of the size of integers that you can work with. Usually, for representing integer data you should just stick to the **int** data type, which is good enough for most purposes.

The **float** data type is represented in four bytes of memory, using a standard method for encoding real numbers. The maximum value for a **float** is about 10 raised to the power 38. A **float** can have about 7 significant digits. (So that 32.3989231134 and 32.3989234399 would both have to be rounded off to about 32.398923 in order to be stored in a variable of type **float**.) A **double** takes up 8 bytes, can range up to about 10 to the power 308, and has about 15 significant digits. Ordinarily, you should stick to the **double** type for real values.

A variable of type **char** occupies two bytes in memory. The value of a **char** variable is a single character such as A, *, x, or a space character. The value can also be a special character such a tab or a carriage return or one of the many Unicode characters that come from different languages. When a character is typed into a program, it must be surrounded by single quotes; for example: `'A'`, `'*'`, or `'x'`. Without the quotes, `A` would be an identifier and `*` would be a multiplication operator. The quotes are not part of the value and are not stored in the variable; they are just a convention for naming a particular character constant in a program.

A name for a constant value is called a ***literal***. A literal is what you have to type in a program to represent a value. `'A'` and `'*'` are literals of type **char**, representing the character values A and *. Certain special characters have special literals that use a backslash, \, as an "escape character". In particular, a tab is represented as `'\t'`, a carriage return as `'\r'`, a linefeed as `'\n'`, the single quote character as `'\''`, and the backslash itself as `'\\'`. Note that even though you type two characters between the quotes in `'\t'`, the value represented by this literal is a single tab character.

Numeric literals are a little more complicated than you might expect. Of course, there are the obvious literals such as 317 and 17.42. But there are other possibilities for expressing numbers in a Java program. First of all, real numbers can be represented in an exponential

form such as 1.3e12 or 12.3737e-108. The "e12" and "e-108" represent powers of 10, so that 1.3e12 means 1.3 times 10^{12} and 12.3737e-108 means 12.3737 times 10^{-108}. This format can be used to express very large and very small numbers. Any numerical literal that contains a decimal point or exponential is a literal of type **double**. To make a literal of type **float**, you have to append an "F" or "f" to the end of the number. For example, "1.2F" stands for 1.2 considered as a value of type **float**. (Occasionally, you need to know this because the rules of Java say that you can't assign a value of type **double** to a variable of type **float**, so you might be confronted with a ridiculous-seeming error message if you try to do something like "`x = 1.2;`" when `x` is a variable of type **float**. You have to say "`x = 1.2F;`". This is one reason why I advise sticking to type **double** for real numbers.)

Even for integer literals, there are some complications. Ordinary integers such as 177777 and -32 are literals of type **byte**, **short**, or **int**, depending on their size. You can make a literal of type **long** by adding "L" as a suffix. For example: 17L or 728476874368L. As another complication, Java allows octal (base-8) and hexadecimal (base-16) literals. I don't want to cover base-8 and base-16 in detail, but in case you run into them in other people's programs, it's worth knowing a few things: Octal numbers use only the digits 0 through 7. In Java, a numeric literal that begins with a 0 is interpreted as an octal number; for example, the literal 045 represents the number 37, not the number 45. Hexadecimal numbers use 16 digits, the usual digits 0 through 9 and the letters A, B, C, D, E, and F. Upper case and lower case letters can be used interchangeably in this context. The letters represent the numbers 10 through 15. In Java, a hexadecimal literal begins with `0x` or `0X`, as in `0x45` or `0xFF7A`.

Hexadecimal numbers are also used in character literals to represent arbitrary Unicode characters. A Unicode literal consists of `\u` followed by four hexadecimal digits. For example, the character literal `'\u00E9'` represents the Unicode character that is an "e" with an acute accent.

Java 7 introduces a couple of minor improvements in numeric literals. First of all, numeric literals in Java 7 can include the underscore character ("_"), which can be used to separate groups of digits. For example, the integer constant for one billion could be written 1_000_000_000, which is a good deal easier to decipher than 1000000000. There is no rule about how many digits have to be in each group. Java 7 also supports binary numbers, using the digits 0 and 1 and the prefix `0b` (or `0B`). For example: `0b10110` or `0b1010_1100_1011`.

For the type **boolean**, there are precisely two literals: `true` and `false`. These literals are typed just as I've written them here, without quotes, but they represent values, not variables. Boolean values occur most often as the values of conditional expressions. For example,

```
rate > 0.05
```

is a boolean-valued expression that evaluates to `true` if the value of the variable `rate` is greater than 0.05, and to `false` if the value of `rate` is not greater than 0.05. As you'll see in Chapter 3, boolean-valued expressions are used extensively in control structures. Of course, boolean values can also be assigned to variables of type **boolean**.

Java has other types in addition to the primitive types, but all the other types represent objects rather than "primitive" data values. For the most part, we are not concerned with objects for the time being. However, there is one predefined object type that is very important: the type *String*. A *String* is a sequence of characters. You've already seen a string literal: `"Hello World!"`. The double quotes are part of the literal; they have to be typed in the program. However, they are not part of the actual string value, which consists of just the characters between the quotes. Within a string, special characters can be represented using the backslash notation. Within this context, the double quote is itself a special character. For

example, to represent the string **value**

```
I said, "Are you listening!"
```

with a linefeed at the end, you would have to type the string **literal**:

```
"I said, \"Are you listening!\"\n"
```

You can also use `\t`, `\r`, `\\`, and Unicode sequences such as `\u00E9` to represent other special characters in string literals. Because strings are objects, their behavior in programs is peculiar in some respects (to someone who is not used to objects). I'll have more to say about them in the next section.

2.2.3 Variables in Programs

A variable can be used in a program only if it has first been ***declared***. A ***variable declaration statement*** is used to declare one or more variables and to give them names. When the computer executes a variable declaration, it sets aside memory for the variable and associates the variable's name with that memory. A simple variable declaration takes the form:

⟨*type-name*⟩ ⟨*variable-name-or-names*⟩;

The ⟨***variable-name-or-names***⟩ can be a single variable name or a list of variable names separated by commas. (We'll see later that variable declaration statements can actually be somewhat more complicated than this.) Good programming style is to declare only one variable in a declaration statement, unless the variables are closely related in some way. For example:

```
int numberOfStudents;
String name;
double x, y;
boolean isFinished;
char firstInitial, middleInitial, lastInitial;
```

It is also good style to include a comment with each variable declaration to explain its purpose in the program, or to give other information that might be useful to a human reader. For example:

```
double principal;    // Amount of money invested.
double interestRate; // Rate as a decimal, not percentage.
```

In this chapter, we will only use variables declared inside the `main()` subroutine of a program. Variables declared inside a subroutine are called ***local variables*** for that subroutine. They exist only inside the subroutine, while it is running, and are completely inaccessible from outside. Variable declarations can occur anywhere inside the subroutine, as long as each variable is declared before it is used in any expression. Some people like to declare all the variables at the beginning of the subroutine. Others like to wait to declare a variable until it is needed. My preference: Declare important variables at the beginning of the subroutine, and use a comment to explain the purpose of each variable. Declare "utility variables" which are not important to the overall logic of the subroutine at the point in the subroutine where they are first used. Here is a simple program using some variables and assignment statements:

```
/**
 * This class implements a simple program that
 * will compute the amount of interest that is
 * earned on $17,000 invested at an interest
 * rate of 0.07 for one year.  The interest and
```

```
 * the value of the investment after one year are
 * printed to standard output.
 */

public class Interest {

   public static void main(String[] args) {

       /* Declare the variables. */

       double principal;     // The value of the investment.
       double rate;          // The annual interest rate.
       double interest;      // Interest earned in one year.

       /* Do the computations. */

       principal = 17000;
       rate = 0.07;
       interest = principal * rate;   // Compute the interest.

       principal = principal + interest;
             // Compute value of investment after one year, with interest.
             // (Note: The new value replaces the old value of principal.)

       /* Output the results. */

       System.out.print("The interest earned is $");
       System.out.println(interest);
       System.out.print("The value of the investment after one year is $");
       System.out.println(principal);

   } // end of main()

} // end of class Interest
```

This program uses several subroutine call statements to display information to the user of the program. Two different subroutines are used: `System.out.print` and `System.out.println`. The difference between these is that `System.out.println` adds a linefeed after the end of the information that it displays, while `System.out.print` does not. Thus, the value of `interest`, which is displayed by the subroutine call "`System.out.println(interest);`", follows on the same line after the string displayed by the previous `System.out.print` statement. Note that the value to be displayed by `System.out.print` or `System.out.println` is provided in parentheses after the subroutine name. This value is called a ***parameter*** to the subroutine. A parameter provides a subroutine with information it needs to perform its task. In a subroutine call statement, any parameters are listed in parentheses after the subroutine name. Not all subroutines have parameters. If there are no parameters in a subroutine call statement, the subroutine name must be followed by an empty pair of parentheses.

All the sample programs for this textbook are available in separate source code files in the on-line version of this text at http://math.hws.edu/javanotes/source. They are also included in the downloadable archives of the web site. The source code for the `Interest` program, for example, can be found in the file *Interest.java*.

2.3 Strings, Objects, Enums, and Subroutines

The previous section introduced the eight primitive data types and the type *String*. There is a fundamental difference between the primitive types and the *String* type: Values of type

String are objects. While we will not study objects in detail until Chapter 5, it will be useful for you to know a little about them and about a closely related topic: classes. This is not just because strings are useful but because objects and classes are essential to understanding another important programming concept, subroutines.

Another reason for considering classes and objects at this point is so that we can introduce ***enums***. An enum is a data type that can be created by a Java programmer to represent a small collection of possible values. Technically, an enum is a class and its possible values are objects. Enums will be our first example of adding a new type to the Java language. We will look at them later in this section.

2.3.1 Built-in Subroutines and Functions

Recall that a subroutine is a set of program instructions that have been chunked together and given a name. In Chapter 4, you'll learn how to write your own subroutines, but you can get a lot done in a program just by calling subroutines that have already been written for you. In Java, every subroutine is contained in a class or in an object. Some classes that are standard parts of the Java language contain predefined subroutines that you can use. A value of type *String*, which is an object, contains subroutines that can be used to manipulate that string. These subroutines are "built into" the Java language. You can call all these subroutines without understanding how they were written or how they work. Indeed, that's the whole point of subroutines: A subroutine is a "black box" which can be used without knowing what goes on inside.

Classes in Java have two very different functions. First of all, a class can group together variables and subroutines that are contained in that class. These variables and subroutines are called ***static members*** of the class. You've seen one example: In a class that defines a program, the `main()` routine is a static member of the class. The parts of a class definition that define static members are marked with the reserved word "`static`", just like the `main()` routine of a program. However, classes have a second function. They are used to describe objects. In this role, the class of an object specifies what subroutines and variables are contained in that object. The class is a **type**—in the technical sense of a specification of a certain type of data value—and the object is a value of that type. For example, *String* is actually the name of a class that is included as a standard part of the Java language. *String* is also a type, and literal strings such as `"Hello World"` represent values of type *String*.

So, every subroutine is contained either in a class or in an object. Classes **contain** subroutines, which are called static member subroutines. Classes also **describe** objects and the subroutines that are contained in those objects.

This dual use can be confusing, and in practice most classes are designed to perform primarily or exclusively in only one of the two possible roles. For example, although the *String* class does contain a few rarely-used static member subroutines, it exists mainly to specify a large number of subroutines that are contained in objects of type *String*. Another standard class, named *Math*, exists entirely to group together a number of static member subroutines that compute various common mathematical functions.

* * *

To begin to get a handle on all of this complexity, let's look at the subroutine `System.out.print` as an example. As you have seen earlier in this chapter, this subroutine is used to display information to the user. For example, `System.out.print("Hello World")` displays the message, Hello World.

System is one of Java's standard classes. One of the static member variables in this class is named `out`. Since this variable is contained in the class *System*, its full name—which you have to use to refer to it in your programs—is `System.out`. The variable `System.out` refers to an object, and that object in turn contains a subroutine named `print`. The compound identifier `System.out.print` refers to the subroutine `print` in the object `out` in the class *System*.

(As an aside, I will note that the object referred to by `System.out` is an object of the class *PrintStream*. *PrintStream* is another class that is a standard part of Java. **Any** object of type *PrintStream* is a destination to which information can be printed; **any** object of type *PrintStream* has a `print` subroutine that can be used to send information to that destination. The object `System.out` is just one possible destination, and `System.out.print` is the subroutine that sends information to that particular destination. Other objects of type *PrintStream* might send information to other destinations such as files or across a network to other computers. This is object-oriented programming: Many different things which have something in common—they can all be used as destinations for information—can all be used in the same way—through a `print` subroutine. The *PrintStream* class expresses the commonalities among all these objects.)

Since class names and variable names are used in similar ways, it might be hard to tell which is which. Remember that all the built-in, predefined names in Java follow the rule that class names begin with an upper case letter while variable names begin with a lower case letter. While this is not a formal syntax rule, I strongly recommend that you follow it in your own programming. Subroutine names should also begin with lower case letters. There is no possibility of confusing a variable with a subroutine, since a subroutine name in a program is always followed by a left parenthesis.

As one final general note, you should be aware that subroutines in Java are often referred to as ***methods***. Generally, the term "method" means a subroutine that is contained in a class or in an object. Since this is true of every subroutine in Java, every subroutine in Java is a method (with one very technical exception). The same is not true for other programming languages. Nevertheless, the term "method" is mostly used in the context of object-oriented programming, and until we start doing real object-oriented programming in Chapter 5, I will prefer to use the more general term, "subroutine." However, I should note that some people prefer to use the term "method" from the beginning.

* * *

Classes can contain static member subroutines, as well as static member variables. For example, the *System* class contains a subroutine named `exit`. In a program, of course, this subroutine must be referred to as `System.exit`. Calling this subroutine will terminate the program. You could use it if you had some reason to terminate the program before the end of the `main` routine. For historical reasons, this subroutine takes an integer as a parameter, so the subroutine call statement might look like "`System.exit(0);`" or "`System.exit(1);`". (The parameter tells the computer why the program was terminated. A parameter value of 0 indicates that the program ended normally. Any other value indicates that the program was terminated because an error was detected. But in practice, the value of the parameter is usually ignored.)

Every subroutine performs some specific task. For some subroutines, that task is to compute or retrieve some data value. Subroutines of this type are called ***functions***. We say that a function ***returns*** a value. Generally, the returned value is meant to be used somehow in the program.

You are familiar with the mathematical function that computes the square root of a number. Java has a corresponding function called `Math.sqrt`. This function is a static member

subroutine of the class named *Math*. If `x` is any numerical value, then `Math.sqrt(x)` computes and returns the square root of that value. Since `Math.sqrt(x)` represents a value, it doesn't make sense to put it on a line by itself in a subroutine call statement such as

```
Math.sqrt(x);   // This doesn't make sense!
```

What, after all, would the computer do with the value computed by the function in this case? You have to tell the computer to do something with the value. You might tell the computer to display it:

```
System.out.print( Math.sqrt(x) );  // Display the square root of x.
```

or you might use an assignment statement to tell the computer to store that value in a variable:

```
lengthOfSide = Math.sqrt(x);
```

The function call `Math.sqrt(x)` represents a value of type **double**, and it can be used anyplace where a numeric literal of type double could be used.

The *Math* class contains many static member functions. Here is a list of some of the more important of them:

- `Math.abs(x)`, which computes the absolute value of `x`.
- The usual trigonometric functions, `Math.sin(x)`, `Math.cos(x)`, and `Math.tan(x)`. (For all the trigonometric functions, angles are measured in radians, not degrees.)
- The inverse trigonometric functions arcsin, arccos, and arctan, which are written as: `Math.asin(x)`, `Math.acos(x)`, and `Math.atan(x)`. The return value is expressed in radians, not degrees.
- The exponential function `Math.exp(x)` for computing the number e raised to the power `x`, and the natural logarithm function `Math.log(x)` for computing the logarithm of `x` in the base e.
- `Math.pow(x,y)` for computing `x` raised to the power `y`.
- `Math.floor(x)`, which rounds `x` down to the nearest integer value that is less than or equal to `x`. Even though the return value is mathematically an integer, it is returned as a value of type **double**, rather than of type **int** as you might expect. For example, `Math.floor(3.76)` is 3.0. The function `Math.round(x)` returns the integer that is closest to `x`.
- `Math.random()`, which returns a randomly chosen **double** in the range `0.0 <= Math.random() < 1.0`. (The computer actually calculates so-called "pseudorandom" numbers, which are not truly random but are random enough for most purposes.)

For these functions, the type of the parameter—the `x` or `y` inside the parentheses—can be any value of any numeric type. For most of the functions, the value returned by the function is of type **double** no matter what the type of the parameter. However, for `Math.abs(x)`, the value returned will be the same type as `x`; if `x` is of type **int**, then so is `Math.abs(x)`. So, for example, while `Math.sqrt(9)` is the **double** value 3.0, `Math.abs(9)` is the **int** value 9.

Note that `Math.random()` does not have any parameter. You still need the parentheses, even though there's nothing between them. The parentheses let the computer know that this is a subroutine rather than a variable. Another example of a subroutine that has no parameters is the function `System.currentTimeMillis()`, from the *System* class. When this function is executed, it retrieves the current time, expressed as the number of milliseconds that have passed since a standardized base time (the start of the year 1970 in Greenwich Mean Time, if you care). One

millisecond is one-thousandth of a second. The return value of `System.currentTimeMillis()` is of type **long** (a 64-bit integer). This function can be used to measure the time that it takes the computer to perform a task. Just record the time at which the task is begun and the time at which it is finished and take the difference.

Here is a sample program that performs a few mathematical tasks and reports the time that it takes for the program to run. On some computers, the time reported might be zero, because it is too small to measure in milliseconds. Even if it's not zero, you can be sure that most of the time reported by the computer was spent doing output or working on tasks other than the program, since the calculations performed in this program occupy only a tiny fraction of a second of a computer's time.

```
/**
 * This program performs some mathematical computations and displays
 * the results.  It then reports the number of seconds that the
 * computer spent on this task.
 */

public class TimedComputation {

   public static void main(String[] args) {

      long startTime; // Starting time of program, in milliseconds.
      long endTime;   // Time when computations are done, in milliseconds.
      double time;    // Time difference, in seconds.

      startTime = System.currentTimeMillis();

      double width, height, hypotenuse;  // sides of a triangle
      width = 42.0;
      height = 17.0;
      hypotenuse = Math.sqrt( width*width + height*height );
      System.out.print("A triangle with sides 42 and 17 has hypotenuse ");
      System.out.println(hypotenuse);

      System.out.println("\nMathematically, sin(x)*sin(x) + "
                                     + "cos(x)*cos(x) - 1 should be 0.");
      System.out.println("Let's check this for x = 1:");
      System.out.print("      sin(1)*sin(1) + cos(1)*cos(1) - 1 is ");
      System.out.println( Math.sin(1)*Math.sin(1)
                                      + Math.cos(1)*Math.cos(1) - 1 );
      System.out.println("(There can be round-off errors when"
                                    + " computing with real numbers!)");

      System.out.print("\nHere is a random number:  ");
      System.out.println( Math.random() );

      endTime = System.currentTimeMillis();
      time = (endTime - startTime) / 1000.0;

      System.out.print("\nRun time in seconds was:  ");
      System.out.println(time);

   } // end main()

} // end class TimedComputation
```

2.3.2 Operations on Strings

A value of type *String* is an object. That object contains data, namely the sequence of characters that make up the string. It also contains subroutines. All of these subroutines are in fact functions. For example, every string object contains a function named `length` that computes the number of characters in that string. Suppose that `advice` is a variable that refers to a *String*. For example, `advice` might have been declared and assigned a value as follows:

```
String advice;
advice = "Seize the day!";
```

Then `advice.length()` is a function call that returns the number of characters in the string "Seize the day!". In this case, the return value would be 14. In general, for any string variable `str`, the value of `str.length()` is an **int** equal to the number of characters in the string that is the value of `str`. Note that this function has no parameter; the particular string whose length is being computed is the value of `str`. The `length` subroutine is defined by the class *String*, and it can be used with any value of type *String*. It can even be used with *String* literals, which are, after all, just constant values of type *String*. For example, you could have a program count the characters in "Hello World" for you by saying

```
System.out.print("The number of characters in ");
System.out.print("the string \"Hello World\" is ");
System.out.println( "Hello World".length() );
```

The *String* class defines a lot of functions. Here are some that you might find useful. Assume that `s1` and `s2` refer to values of type *String*:

- `s1.equals(s2)` is a function that returns a **boolean** value. It returns `true` if `s1` consists of exactly the same sequence of characters as `s2`, and returns `false` otherwise.
- `s1.equalsIgnoreCase(s2)` is another boolean-valued function that checks whether `s1` is the same string as `s2`, but this function considers upper and lower case letters to be equivalent. Thus, if `s1` is "cat", then `s1.equals("Cat")` is `false`, while `s1.equalsIgnoreCase("Cat")` is `true`.
- `s1.length()`, as mentioned above, is an integer-valued function that gives the number of characters in `s1`.
- `s1.charAt(N)`, where `N` is an integer, returns a value of type **char**. It returns the `N`-th character in the string. Positions are numbered starting with 0, so `s1.charAt(0)` is actually the first character, `s1.charAt(1)` is the second, and so on. The final position is `s1.length() - 1`. For example, the value of `"cat".charAt(1)` is 'a'. An error occurs if the value of the parameter is less than zero or greater than `s1.length() - 1`.
- `s1.substring(N,M)`, where `N` and `M` are integers, returns a value of type *String*. The returned value consists of the characters of `s1` in positions `N`, `N+1`,..., `M-1`. Note that the character in position `M` is not included. The returned value is called a substring of `s1`. The subroutine `s1.substring(N)` returns the substring of `s1` consisting of characters starting at position `N` up until the end of the string.
- `s1.indexOf(s2)` returns an integer. If `s2` occurs as a substring of `s1`, then the returned value is the starting position of that substring. Otherwise, the returned value is -1. You can also use `s1.indexOf(ch)` to search for a particular character, `ch`, in `s1`. To find the first occurrence of `x` at or after position `N`, you can use `s1.indexOf(x,N)`.

- `s1.compareTo(s2)` is an integer-valued function that compares the two strings. If the strings are equal, the value returned is zero. If `s1` is less than `s2`, the value returned is a number less than zero, and if `s1` is greater than `s2`, the value returned is some number greater than zero. (If both of the strings consist entirely of lower case letters, or if they consist entirely of upper case letters, then "less than" and "greater than" refer to alphabetical order. Otherwise, the ordering is more complicated.)
- `s1.toUpperCase()` is a *String*-valued function that returns a new string that is equal to `s1`, except that any lower case letters in `s1` have been converted to upper case. For example, `"Cat".toUpperCase()` is the string `"CAT"`. There is also a function `s1.toLowerCase()`.
- `s1.trim()` is a *String*-valued function that returns a new string that is equal to `s1` except that any non-printing characters such as spaces and tabs have been trimmed from the beginning and from the end of the string. Thus, if `s1` has the value `"fred  "`, then `s1.trim()` is the string `"fred"`, with the spaces at the end removed.

For the functions `s1.toUpperCase()`, `s1.toLowerCase()`, and `s1.trim()`, note that the value of `s1` is **not** modified. Instead a new string is created and returned as the value of the function. The returned value could be used, for example, in an assignment statement such as "`smallLetters = s1.toLowerCase();`". To change the value of `s1`, you could use an assignment "`s1 = s1.toLowerCase();`".

* * *

Here is another extremely useful fact about strings: You can use the plus operator, `+`, to ***concatenate*** two strings. The concatenation of two strings is a new string consisting of all the characters of the first string followed by all the characters of the second string. For example, `"Hello" + "World"` evaluates to `"HelloWorld"`. (Gotta watch those spaces, of course—if you want a space in the concatenated string, it has to be somewhere in the input data, as in `"Hello " + "World"`.)

Let's suppose that `name` is a variable of type *String* and that it already refers to the name of the person using the program. Then, the program could greet the user by executing the statement:

```
System.out.println("Hello, "  +  name  +  ".  Pleased to meet you!");
```

Even more surprising is that you can actually concatenate values of **any** type onto a *String* using the `+` operator. The value is converted to a string, just as it would be if you printed it to the standard output, and then it is concatenated onto the string. For example, the expression `"Number" + 42` evaluates to the string `"Number42"`. And the statements

```
System.out.print("After ");
System.out.print(years);
System.out.print(" years, the value is ");
System.out.print(principal);
```

can be replaced by the single statement:

```
System.out.print("After " + years +
                    " years, the value is " + principal);
```

Obviously, this is very convenient. It would have shortened some of the examples presented earlier in this chapter.

2.3.3 Introduction to Enums

Java comes with eight built-in primitive types and a large set of types that are defined by classes, such as *String*. But even this large collection of types is not sufficient to cover all the possible situations that a programmer might have to deal with. So, an essential part of Java, just like almost any other programming language, is the ability to create **new** types. For the most part, this is done by defining new classes; you will learn how to do that in Chapter 5. But we will look here at one particular case: the ability to define ***enums*** (short for ***enumerated types***). Enums are a recent addition to Java. They were only added in Version 5.0. Many programming languages have something similar, and many people believe that enums should have been part of Java from the beginning.

Technically, an enum is considered to be a special kind of class, but that is not important for now. In this section, we will look at enums in a simplified form. In practice, most uses of enums will only need the simplified form that is presented here.

An enum is a type that has a fixed list of possible values, which is specified when the enum is created. In some ways, an enum is similar to the **boolean** data type, which has `true` and `false` as its only possible values. However, **boolean** is a primitive type, while an enum is not.

The definition of an enum type has the (simplified) form:

```
enum ⟨enum-type-name⟩ { ⟨list-of-enum-values⟩ }
```

This definition cannot be inside a subroutine. You can place it **outside** the `main()` routine of the program. The ⟨*enum-type-name*⟩ can be any simple identifier. This identifier becomes the name of the enum type, in the same way that "boolean" is the name of the **boolean** type and "String" is the name of the *String* type. Each value in the ⟨*list-of-enum-values*⟩ must be a simple identifier, and the identifiers in the list are separated by commas. For example, here is the definition of an enum type named *Season* whose values are the names of the four seasons of the year:

```
enum Season { SPRING, SUMMER, FALL, WINTER }
```

By convention, enum values are given names that are made up of upper case letters, but that is a style guideline and not a syntax rule. Enum values are not variables. Each value is a ***constant*** that always has the same value. In fact, the possible values of an enum type are usually referred to as ***enum constants***.

Note that the enum constants of type *Season* are considered to be "contained in" *Season*, which means—following the convention that compound identifiers are used for things that are contained in other things—the names that you actually use in your program to refer to them are `Season.SPRING`, `Season.SUMMER`, `Season.FALL`, and `Season.WINTER`.

Once an enum type has been created, it can be used to declare variables in exactly the same ways that other types are used. For example, you can declare a variable named `vacation` of type `Season` with the statement:

```
Season vacation;
```

After declaring the variable, you can assign a value to it using an assignment statement. The value on the right-hand side of the assignment can be one of the enum constants of type `Season`. Remember to use the full name of the constant, including "Season"! For example:

```
vacation = Season.SUMMER;
```

You can print out an enum value with an output statement such as `System.out.print(vacation)`. The output value will be the name of the enum constant (without the "Season."). In this case, the output would be "SUMMER".

Because an enum is technically a class, the enum values are technically objects. As objects, they can contain subroutines. One of the subroutines in every enum value is named `ordinal()`. When used with an enum value, it returns the ***ordinal number*** of the value in the list of values of the enum. The ordinal number simply tells the position of the value in the list. That is, `Season.SPRING.ordinal()` is the **int** value 0, `Season.SUMMER.ordinal()` is 1, `Season.FALL.ordinal()` is 2, and `Season.WINTER.ordinal()` is 3. (You will see over and over again that computer scientists like to start counting at zero!) You can, of course, use the `ordinal()` method with a variable of type *Season*, such as `vacation.ordinal()` in our example.

Right now, it might not seem to you that enums are all that useful. As you work though the rest of the book, you should be convinced that they are. For now, you should at least appreciate them as the first example of an important concept: creating new types. Here is a little example that shows enums being used in a complete program:

```
public class EnumDemo {

       // Define two enum types -- remember that the definitions
       // go OUTSIDE The main() routine!

    enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY }

    enum Month { JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC }

    public static void main(String[] args) {

         Day tgif;     // Declare a variable of type Day.
         Month libra;  // Declare a variable of type Month.

         tgif = Day.FRIDAY;    // Assign a value of type Day to tgif.
         libra = Month.OCT;    // Assign a value of type Month to libra.

         System.out.print("My sign is libra, since I was born in ");
         System.out.println(libra);   // Output value will be:  OCT
         System.out.print("That's the ");
         System.out.print( libra.ordinal() );
         System.out.println("-th month of the year.");
         System.out.println("   (Counting from 0, of course!)");

         System.out.print("Isn't it nice to get to ");
         System.out.println(tgif);   // Output value will be:  FRIDAY

         System.out.println( tgif + " is the " + tgif.ordinal()
                                          + "-th day of the week.");
              // You can concatenate enum values onto Strings!

    }

}
```

2.4 Text Input and Output

For some unfathomable reason, Java has never made it very easy to read data typed in by the user of a program. You've already seen that output can be displayed to the user using the subroutine `System.out.print`. This subroutine is part of a pre-defined object called `System.out`. The purpose of this object is precisely to display output to the user. There is

a corresponding object called `System.in` that exists to read data input by the user, but it provides only very primitive input facilities, and it requires some advanced Java programming skills to use it effectively.

Java 5.0 finally made input from any source a little easier with a new *Scanner* class. However, it requires some knowledge of object-oriented programming to use this class, so it's not appropriate for use here at the beginning of this course. Java 6 introduced the *Console* class, specifically for communicating with the user, but again, using *Console* requires more knowledge about objects than you have at this point. (Furthermore, in my opinion, *Scanner* and *Console* still don't get things quite right. Nevertheless, I will introduce *Scanner* briefly at the end of this section, in case you want to start using it now.)

There is some excuse for this lack of concern with input, since Java is meant mainly to write programs for Graphical User Interfaces, and those programs have their own style of input/output, which **is** implemented quite well in Java. However, basic support is needed for input/output in old-fashioned non-GUI programs. Fortunately, it is possible to **extend** Java by creating new classes that provide subroutines that are not available in the standard part of the language. As soon as a new class is available, the subroutines that it contains can be used in exactly the same way as built-in routines.

Along these lines, I've written a class called *TextIO* that defines subroutines for reading values typed by the user of a non-GUI program. The subroutines in this class make it possible to get input from the standard input object, `System.in`, without knowing about the advanced aspects of Java that are needed to use *Scanner* or to use `System.in` directly. *TextIO* also contains a set of output subroutines. The output subroutines are similar to those provided in `System.out`, but they provide a few additional features. For displaying output to the user, you can use either `System.out` or `TextIO`, and you can even mix them in the same program.

To use the *TextIO* class, you must make sure that the class is available to your program. What this means depends on the Java programming environment that you are using. In general, you just have to add the source code file, *TextIO.java*, to the same directory that contains your main program. See Section 2.6 for more information about how to use *TextIO*.

2.4.1 A First Text Input Example

The input routines in the *TextIO* class are static member functions. (Static member functions were introduced in the previous section.) Let's suppose that you want your program to read an integer typed in by the user. The *TextIO* class contains a static member function named `getlnInt` that you can use for this purpose. Since this function is contained in the *TextIO* class, you have to refer to it in your program as `TextIO.getlnInt`. The function has no parameters, so a complete call to the function takes the form "`TextIO.getlnInt()`". This function call represents the **int** value typed by the user, and you have to do something with the returned value, such as assign it to a variable. For example, if `userInput` is a variable of type **int** (created with a declaration statement "`int userInput;`"), then you could use the assignment statement

```
userInput = TextIO.getlnInt();
```

When the computer executes this statement, it will wait for the user to type in an integer value. That value will then be returned by the function, and it will be stored in the variable, `userInput`. Here is a complete program that uses `TextIO.getlnInt` to read a number typed by the user and then prints out the square of the number that the user types:

```
/**
 * A program that reads an integer that is typed in by the
 * user and computes and prints the square of that integer.
 */

public class PrintSquare {

     public static void main(String[] args) {

        int userInput;  // The number input by the user.
        int square;     // The userInput, multiplied by itself.

        System.out.print("Please type a number: ");
        userInput = TextIO.getlnInt();
        square = userInput * userInput;
        System.out.print("The square of that number is ");
        System.out.println(square);

     } // end of main()

} //end of class PrintSquare
```

When you run this program, it will display the message "Please type a number:" and will pause until you type a response, including a carriage return after the number.

2.4.2 Text Output

The *TextIO* class contains static member subroutines `TextIO.put` and `TextIO.putln` that can be used in the same way as `System.out.print` and `System.out.println`. For example, although there is no particular advantage in doing so in this case, you could replace the two lines

```
System.out.print("The square of that number is ");
System.out.println(square);
```

with

```
TextIO.put("The square of that number is ");
TextIO.putln(square);
```

For the next few chapters, I will use *TextIO* for input in all my examples, and I will often use it for output. Keep in mind that *TextIO* can only be used in a program if it is available to that program. It is not built into Java in the way that the *System* class is.

Let's look a little more closely at the built-in output subroutines `System.out.print` and `System.out.println`. Each of these subroutines can be used with one parameter, where the parameter can be a value of any of the primitive types **byte**, **short**, **int**, **long**, **float**, **double**, **char**, or **boolean**. The parameter can also be a *String*, a value belonging to an enum type, or indeed any object. That is, you can say "`System.out.print(x);`" or "`System.out.println(x);`", where `x` is any expression whose value is of any type whatsoever. The expression can be a constant, a variable, or even something more complicated such as `2*distance*time`. Now, in fact, the *System* class actually includes several different subroutines to handle different parameter types. There is one `System.out.print` for printing values of type **double**, one for values of type **int**, another for values that are objects, and so on. These subroutines can have the same name since the computer can tell which one you mean in a given subroutine call statement, depending on the type of parameter that you supply. Having several subroutines of the same

name that differ in the types of their parameters is called ***overloading***. Many programming languages do not permit overloading, but it is common in Java programs.

The difference between `System.out.print` and `System.out.println` is that the `println` version outputs a carriage return after it outputs the specified parameter value. There is a version of `System.out.println` that has no parameters. This version simply outputs a carriage return, and nothing else. A subroutine call statement for this version of the subroutine looks like "`System.out.println();`", with empty parentheses. Note that "`System.out.println(x);`" is exactly equivalent to "`System.out.print(x); System.out.println();`"; the carriage return comes **after** the value of `x`. (There is no version of `System.out.print` without parameters. Do you see why?)

As mentioned above, the *TextIO* subroutines `TextIO.put` and `TextIO.putln` can be used as replacements for `System.out.print` and `System.out.println`. The *TextIO* functions work in exactly the same way as the *System* functions, except that, as we will see below, *TextIO* can also be used to write to other destinations.

2.4.3 TextIO Input Functions

The *TextIO* class is a little more versatile at doing output than is `System.out`. However, it's input for which we really need it.

With *TextIO*, input is done using functions. For example, `TextIO.getlnInt()`, which was discussed above, makes the user type in a value of type **int** and returns that input value so that you can use it in your program. *TextIO* includes several functions for reading different types of input values. Here are examples of the ones that you are most likely to use:

```
j = TextIO.getlnInt();     // Reads a value of type int.
y = TextIO.getlnDouble();  // Reads a value of type double.
a = TextIO.getlnBoolean(); // Reads a value of type boolean.
c = TextIO.getlnChar();    // Reads a value of type char.
w = TextIO.getlnWord();    // Reads one "word" as a value of type String.
s = TextIO.getln();        // Reads an entire input line as a String.
```

For these statements to be legal, the variables on the left side of each assignment statement must already be declared and must be of the same type as that returned by the function on the right side. Note carefully that these functions do not have parameters. The values that they return come from outside the program, typed in by the user as the program is running. To "capture" that data so that you can use it in your program, you have to assign the return value of the function to a variable. You will then be able to refer to the user's input value by using the name of the variable.

When you call one of these functions, you are guaranteed that it will return a legal value of the correct type. If the user types in an illegal value as input—for example, if you ask for an **int** and the user types in a non-numeric character or a number that is outside the legal range of values that can be stored in a variable of type **int**—then the computer will ask the user to re-enter the value, and your program never sees the first, illegal value that the user entered. For `TextIO.getlnBoolean()`, the user is allowed to type in any of the following: true, false, t, f, yes, no, y, n, 1, or 0. Furthermore, they can use either upper or lower case letters. In any case, the user's input is interpreted as a true/false value. It's convenient to use `TextIO.getlnBoolean()` to read the user's response to a Yes/No question.

You'll notice that there are two input functions that return Strings. The first, `getlnWord()`, returns a string consisting of non-blank characters only. When it is called, it skips over any spaces and carriage returns typed in by the user. Then it reads non-blank characters until it gets

to the next space or carriage return. It returns a *String* consisting of all the non-blank characters that it has read. The second input function, `getln()`, simply returns a string consisting of all the characters typed in by the user, including spaces, up to the next carriage return. It gets an entire line of input text. The carriage return itself is not returned as part of the input string, but it is read and discarded by the computer. Note that the String returned by this function might be the ***empty string***, `""`, which contains no characters at all. You will get this return value if the user simply presses return, without typing anything else first.

All the other input functions listed—`getlnInt()`, `getlnDouble()`, `getlnBoolean()`, and `getlnChar()`—behave like `getlnWord()` in that they will skip past any blanks and carriage returns in the input before reading a value.

Furthermore, if the user types extra characters on the line after the input value, **all the extra characters will be discarded, along with the carriage return at the end of the line**. If the program executes another input function, the user will have to type in another line of input. It might not sound like a good idea to discard any of the user's input, but it turns out to be the safest thing to do in most programs. Sometimes, however, you do want to read more than one value from the same line of input. *TextIO* provides the following alternative input functions to allow you to do this:

```
j = TextIO.getInt();     // Reads a value of type int.
y = TextIO.getDouble();  // Reads a value of type double.
a = TextIO.getBoolean(); // Reads a value of type boolean.
c = TextIO.getChar();    // Reads a value of type char.
w = TextIO.getWord();    // Reads one "word" as a value of type String.
```

The names of these functions start with "get" instead of "getln". "Getln" is short for "get line" and should remind you that the functions whose names begin with "getln" will get an entire line of data. A function without the "ln" will read an input value in the same way, but will then save the rest of the input line in a chunk of internal memory called the ***input buffer***. The next time the computer wants to read an input value, it will look in the input buffer before prompting the user for input. This allows the computer to read several values from one line of the user's input. Strictly speaking, the computer actually reads **only** from the input buffer. The first time the program tries to read input from the user, the computer will wait while the user types in an entire line of input. *TextIO* stores that line in the input buffer until the data on the line has been read or discarded (by one of the "getln" functions). The user only gets to type when the buffer is empty.

Clearly, the semantics of input is much more complicated than the semantics of output! Fortunately, for the majority of applications, it's pretty straightforward in practice. You only need to follow the details if you want to do something fancy. In particular, I **strongly** advise you to use the "getln" versions of the input routines, rather than the "get" versions, unless you really want to read several items from the same line of input, precisely because the semantics of the "getln" versions is much simpler.

Note, by the way, that although the *TextIO* input functions will skip past blank spaces and carriage returns while looking for input, they will **not** skip past other characters. For example, if you try to read two **int**s and the user types "2,3", the computer will read the first number correctly, but when it tries to read the second number, it will see the comma. It will regard this as an error and will force the user to retype the number. If you want to input several numbers from one line, you should make sure that the user knows to separate them with spaces, not commas. Alternatively, if you want to require a comma between the numbers, use `getChar()` to read the comma before reading the second number.

There is another character input function, `TextIO.getAnyChar()`, which does not skip past blanks or carriage returns. It simply reads and returns the next character typed by the user, even if it's a blank or carriage return. If the user typed a carriage return, then the **char** returned by `getAnyChar()` is the special linefeed character '\n'. There is also a function, `TextIO.peek()`, that lets you look ahead at the next character in the input without actually reading it. After you "peek" at the next character, it will still be there when you read the next item from input. This allows you to look ahead and see what's coming up in the input, so that you can take different actions depending on what's there.

The *TextIO* class provides a number of other functions. To learn more about them, you can look at the comments in the source code file, *TextIO.java*.

(You might be wondering why there are only two output routines, `print` and `println`, which can output data values of any type, while there is a separate input routine for each data type. As noted above, in reality there are many `print` and `println` routines, one for each data type. The computer can tell them apart based on the type of the parameter that you provide. However, the input routines don't have parameters, so the different input routines can only be distinguished by having different names.)

* * *

Using *TextIO* for input and output, we can now improve the program from Section 2.2 for computing the value of an investment. We can have the user type in the initial value of the investment and the interest rate. The result is a much more useful program—for one thing, it makes sense to run it more than once!

```
/**
 * This class implements a simple program that will compute
 * the amount of interest that is earned on an investment over
 * a period of one year.  The initial amount of the investment
 * and the interest rate are input by the user.  The value of
 * the investment at the end of the year is output.  The
 * rate must be input as a decimal, not a percentage (for
 * example, 0.05 rather than 5).
 */

public class Interest2 {

   public static void main(String[] args) {

       double principal;  // The value of the investment.
       double rate;       // The annual interest rate.
       double interest;   // The interest earned during the year.

       TextIO.put("Enter the initial investment: ");
       principal = TextIO.getlnDouble();

       TextIO.put("Enter the annual interest rate (decimal, not percentage!): ");
       rate = TextIO.getlnDouble();

       interest = principal * rate;       // Compute this year's interest.
       principal = principal + interest;  // Add it to principal.

       TextIO.put("The value of the investment after one year is $");
       TextIO.putln(principal);

   } // end of main()

} // end of class Interest2
```

2.4.4 Formatted Output

If you ran the preceding `Interest2` example, you might have noticed that the answer is not always written in the format that is usually used for dollar amounts. In general, dollar amounts are written with two digits after the decimal point. But the program's output can be a number like 1050.0 or 43.575. It would be better if these numbers were printed as 1050.00 and 43.58.

Java 5.0 introduced a formatted output capability that makes it much easier than it used to be to control the format of output numbers. A lot of formatting options are available. I will cover just a few of the simplest and most commonly used possibilities here.

You can use the function `System.out.printf` to produce formatted output. (The name "printf," which stands for "print formatted," is copied from the C and C++ programming languages, which have always had a similar formatting capability). `System.out.printf` takes two or more parameters. The first parameter is a *String* that specifies the format of the output. This parameter is called the ***format string***. The remaining parameters specify the values that are to be output. Here is a statement that will print a number in the proper format for a dollar amount, where `amount` is a variable of type **double**:

```
System.out.printf( "%1.2f", amount );
```

TextIO can also do formatted output. The function `TextIO.putf` has the same functionality as `System.out.printf`. Using *TextIO*, the above example would be: `TextIO.putf("%1.2f",amount);` and you could say `TextIO.putf("%1.2f",principal);` instead of `TextIO.putln(principal);` in the `Interest2` program to get the output in the right format.

The output format of a value is specified by a ***format specifier***. The format string (in the simple cases that I cover here) contains one format specifier for each of the values that is to be output. Some typical format specifiers are `%d`, `%12d`, `%10s`, `%1.2f`, `%15.8e` and `%1.8g`. Every format specifier begins with a percent sign (`%`) and ends with a letter, possibly with some extra formatting information in between. The letter specifies the type of output that is to be produced. For example, in `%d` and `%12d`, the "d" specifies that an integer is to be written. The "12" in `%12d` specifies the minimum number of spaces that should be used for the output. If the integer that is being output takes up fewer than 12 spaces, extra blank spaces are added in front of the integer to bring the total up to 12. We say that the output is "right-justified in a field of length 12." The value is not forced into 12 spaces; if the value has more than 12 digits, all the digits will be printed, with no extra spaces. The specifier `%d` means the same as `%1d`—that is, an integer will be printed using just as many spaces as necessary. (The "d," by the way, stands for "decimal"—that is, base-10—numbers. You can replace the "d" with an "x" to output an integer value in hexadecimal form.)

The letter "s" at the end of a format specifier can be used with any type of value. It means that the value should be output in its default format, just as it would be in unformatted output. A number, such as the "10" in `%10s` can be added to specify the (minimum) number of characters. The "s" stands for "string," meaning that the value is converted into a *String* value in the usual way.

The format specifiers for values of type **double** are even more complicated. An "f", as in `%1.2f`, is used to output a number in "floating-point" form, that is with digits after the decimal point. In `%1.2f`, the "2" specifies the number of digits to use after the decimal point. The "1" specifies the (minimum) number of characters to output, which effectively means that just as many characters as are necessary should be used. Similarly, `%12.3f` would specify a floating-point format with 3 digits after the decimal point, right-justified in a field of length 12.

Very large and very small numbers should be written in exponential format, such as 6.00221415e23, representing "6.00221415 times 10 raised to the power 23." A format specifier such as `%15.8e` specifies an output in exponential form, with the "8" telling how many digits to use after the decimal point. If you use "g" instead of "e", the output will be in floating-point form for small values and in exponential form for large values. In `%1.8g`, the 8 gives the total number of digits in the answer, including both the digits before the decimal point and the digits after the decimal point.

For numeric output, the format specifier can include a comma (","), which will cause the digits of the number to be separated into groups, to make it easier to read big numbers. In the United States, groups of three digits are separated by commas. For example, if `x` is one billion, then `System.out.printf("%,d",x)` will output 1,000,000,000. In other countries, the separator character and the number of digits per group might be different. The comma should come at the beginning of the format specifier, before the field width; for example: `%,12.3f`.

In addition to format specifiers, the format string in a `printf` statement can include other characters. These extra characters are just copied to the output. This can be a convenient way to insert values into the middle of an output string. For example, if `x` and `y` are variables of type **int**, you could say

```
System.out.printf("The product of %d and %d is %d", x, y, x*y);
```

When this statement is executed, the value of `x` is substituted for the first `%d` in the string, the value of `y` for the second `%d`, and the value of the expression `x*y` for the third, so the output would be something like "The product of 17 and 42 is 714" (quotation marks not included in output!).

2.4.5 Introduction to File I/O

`System.out` sends its output to the output destination known as "standard output." But standard output is just one possible output destination. For example, data can be written to a ***file*** that is stored on the user's hard drive. The advantage to this, of course, is that the data is saved in the file even after the program ends, and the user can print the file, email it to someone else, edit it with another program, and so on.

TextIO has the ability to write data to files and to read data from files. When you write output using the `put`, `putln`, or `putf` method in *TextIO*, the output is sent to the ***current output destination.*** By default, the current output destination is standard output. However, *TextIO* has some subroutines that can be used to change the current output destination. To write to a file named "result.txt", for example, you would use the statement:

```
TextIO.writeFile("result.txt");
```

After this statement is executed, any output from *TextIO* output statements will be sent to the file named "result.txt" instead of to standard output. The file should be created in the same directory that contains the program. Note that if a file with the same name already exists, its previous contents will be erased! In many cases, you want to let the user select the file that will be used for output. The statement

```
TextIO.writeUserSelectedFile();
```

will open a typical graphical-user-interface file selection dialog where the user can specify the output file. If you want to go back to sending output to standard output, you can say

```
TextIO.writeStandardOutput();
```

You can also specify the input source for *TextIO's* various "get" functions. The default input source is standard input. You can use the statement `TextIO.readFile("data.txt")` to read from a file named "data.txt" instead, or you can let the user select the input file by saying `TextIO.readUserSelectedFile()`. You can go back to reading from standard input with `TextIO.readStandardInput()`.

When your program is reading from standard input, the user gets a chance to correct any errors in the input. This is not possible when the program is reading from a file. If illegal data is found when a program tries to read from a file, an error occurs that will crash the program. (Later, we will see that it is possible to "catch" such errors and recover from them.) Errors can also occur, though more rarely, when writing to files.

A complete understanding of file input/output in Java requires a knowledge of object oriented programming. We will return to the topic later, in Chapter 11. The file I/O capabilities in *TextIO* are rather primitive by comparison. Nevertheless, they are sufficient for many applications, and they will allow you to get some experience with files sooner rather than later.

As a simple example, here is a program that asks the user some questions and outputs the user's responses to a file named "profile.txt":

```
public class CreateProfile {

    public static void main(String[] args) {

        String name;     // The user's name.
        String email;    // The user's email address.
        double salary;   // the user's yearly salary.
        String favColor; // The user's favorite color.

        TextIO.putln("Good Afternoon!  This program will create");
        TextIO.putln("your profile file, if you will just answer");
        TextIO.putln("a few simple questions.");
        TextIO.putln();

        /* Gather responses from the user. */

        TextIO.put("What is your name?           ");
        name = TextIO.getln();
        TextIO.put("What is your email address?  ");
        email = TextIO.getln();
        TextIO.put("What is your yearly income?  ");
        salary = TextIO.getlnDouble();
        TextIO.put("What is your favorite color? ");
        favColor = TextIO.getln();

        /* Write the user's information to the file named profile.txt. */

        TextIO.writeFile("profile.txt");  // subsequent output goes to the file
        TextIO.putln("Name:            " + name);
        TextIO.putln("Email:           " + email);
        TextIO.putln("Favorite Color:  " + favColor);
        TextIO.putf( "Yearly Income:   %,1.2f\n", salary);
                   // The "/n" in the previous line is a carriage return, and the
                   // comma in %,1.2f adds separators between groups of digits.

        /* Print a final message to standard output. */

        TextIO.writeStandardOutput();
        TextIO.putln("Thank you.  Your profile has been written to profile.txt.");
```

```
    }
}
```

2.4.6 Using Scanner for Input

TextIO makes it easy to get input from the user. However, since it is not a standard class, you have to remember to add `TextIO.java` to a program that uses it. One advantage of using the *Scanner* class for input is that it's a standard part of Java and so is always there when you want it.

It's not that hard to use a *Scanner* for user input, but doing so requires some syntax that will not be introduced until Chapter 4 and Chapter 5. I'll tell you how to do it here, without explaining why it works. You won't understand all the syntax at this point. (*Scanners* will be covered in more detail in Subsection 11.1.5.)

First, you should add the following line to your program at the beginning of the source code file, **before** the "public class...":

```
import java.util.Scanner;
```

Then include the following statement at the beginning of your `main()` routine:

```
Scanner stdin = new Scanner( System.in );
```

This creates a variable named `stdin` of type *Scanner*. (You can use a different name for the variable if you want; "stdin" stands for "standard input.") You can then use `stdin` in your program to access a variety of subroutines for reading user input. For example, the function `stdin.nextInt()` reads one value of type **int** from the user and returns it. It is almost the same as `TextIO.getInt()` except for two things: If the value entered by the user is not a legal **int**, then `stdin.nextInt()` will crash rather than prompt the user to re-enter the value. And the integer entered by the user must be followed by a blank space or by an end-of-line, whereas `TextIO.getInt()` will stop reading at any character that is not a digit.

There are corresponding methods for reading other types of data, including `stdin.nextDouble()`, `stdin.nextLong()`, and `stdin.nextBoolean()`. (`stdin.nextBoolean()` will only accept "true" or "false" as input.) The method `stdin.nextLine()` is equivalent to `TextIO.getln()`, and `stdin.next()`, like `TextIO.getWord()`, returns a string of non-blank characters.

As a simple example, here is a version of the sample program *Interest2.java* that uses *Scanner* instead of *TextIO* for user input:

```
import java.util.Scanner;  // Make the Scanner class available.

public class Interest2WithScanner {

   public static void main(String[] args) {

      Scanner stdin = new Scanner( System.in );  // Create the Scanner.

      double principal;  // The value of the investment.
      double rate;       // The annual interest rate.
      double interest;   // The interest earned during the year.

      System.out.print("Enter the initial investment: ");
      principal = stdin.nextDouble();

      System.out.print("Enter the annual interest rate (decimal, not percent!): ");
```

```
        rate = stdin.nextDouble();

        interest = principal * rate;       // Compute this year's interest.
        principal = principal + interest;  // Add it to principal.

        System.out.print("The value of the investment after one year is $");
        System.out.println(principal);

    } // end of main()

} // end of class Interest2With Scanner
```

Note the inclusion of the two lines given above and the substitution of `stdin.nextDouble()` for `TextIO.getlnDouble()`. (In fact, `stdin.nextDouble()` is really equivalent to `TextIO.getDouble()` rather than to the "getln" version, but this will not affect the behavior of the program as long as the user types just one number on each line of input.)

I will continue to use *TextIO* for input for the time being, but I will give a few more examples of using *Scanner* in the on-line solutions to the end-of-chapter exercises. There will be more detailed coverage of *Scanner* later in the book.

2.5 Details of Expressions

This section takes a closer look at expressions. Recall that an expression is a piece of program code that represents or computes a value. An expression can be a literal, a variable, a function call, or several of these things combined with operators such as `+` and `>`. The value of an expression can be assigned to a variable, used as a parameter in a subroutine call, or combined with other values into a more complicated expression. (The value can even, in some cases, be ignored, if that's what you want to do; this is more common than you might think.) Expressions are an essential part of programming. So far, these notes have dealt only informally with expressions. This section tells you the more-or-less complete story (leaving out some of the less commonly used operators).

The basic building blocks of expressions are literals (such as `674`, `3.14`, `true`, and `'X'`), variables, and function calls. Recall that a function is a subroutine that returns a value. You've already seen some examples of functions, such as the input routines from the *TextIO* class and the mathematical functions from the *Math* class.

The *Math* class also contains a couple of mathematical constants that are useful in mathematical expressions: `Math.PI` represents π (the ratio of the circumference of a circle to its diameter), and `Math.E` represents e (the base of the natural logarithms). These "constants" are actually member variables in *Math* of type **double**. They are only approximations for the mathematical constants, which would require an infinite number of digits to specify exactly.

Literals, variables, and function calls are simple expressions. More complex expressions can be built up by using ***operators*** to combine simpler expressions. Operators include `+` for adding two numbers, `>` for comparing two values, and so on. When several operators appear in an expression, there is a question of ***precedence***, which determines how the operators are grouped for evaluation. For example, in the expression "`A + B * C`", `B*C` is computed first and then the result is added to `A`. We say that multiplication (`*`) has ***higher precedence*** than addition (`+`). If the default precedence is not what you want, you can use parentheses to explicitly specify the grouping you want. For example, you could use "`(A + B) * C`" if you want to add `A` to `B` first and then multiply the result by `C`.

The rest of this section gives details of operators in Java. The number of operators in Java is quite large, and I will not cover them all here. Most of the important ones are here; a few will be covered in later chapters as they become relevant.

2.5.1 Arithmetic Operators

Arithmetic operators include addition, subtraction, multiplication, and division. They are indicated by `+`, `-`, `*`, and `/`. These operations can be used on values of any numeric type: **byte**, **short**, **int**, **long**, **float**, or **double**. (They can also be used with values of type **char**, which are treated as integers in this context; a **char** is converted into its Unicode code number when it is used with an arithmetic operator.) When the computer actually calculates one of these operations, the two values that it combines must be of the same type. If your program tells the computer to combine two values of different types, the computer will convert one of the values from one type to another. For example, to compute 37.4 + 10, the computer will convert the integer 10 to a real number 10.0 and will then compute 37.4 + 10.0. This is called a ***type conversion***. Ordinarily, you don't have to worry about type conversion in expressions, because the computer does it automatically.

When two numerical values are combined (after doing type conversion on one of them, if necessary), the answer will be of the same type. If you multiply two **int**s, you get an **int**; if you multiply two **double**s, you get a **double**. This is what you would expect, but you have to be very careful when you use the division operator `/`. When you divide two integers, the answer will always be an integer; if the quotient has a fractional part, it is discarded. For example, the value of `7/2` is `3`, not `3.5`. If `N` is an integer variable, then `N/100` is an integer, and `1/N` is equal to zero for any `N` greater than one! This fact is a common source of programming errors. You can force the computer to compute a real number as the answer by making one of the operands real: For example, when the computer evaluates `1.0/N`, it first converts `N` to a real number in order to match the type of `1.0`, so you get a real number as the answer.

Java also has an operator for computing the remainder when one integer is divided by another. This operator is indicated by `%`. If `A` and `B` are integers, then `A % B` represents the remainder when `A` is divided by `B`. (However, for negative operands, `%` is not quite the same as the usual mathematical "modulus" operator, since if one of `A` or `B` is negative, then the value of `A % B` will be negative.) For example, `7 % 2` is `1`, while `34577 % 100` is `77`, and `50 % 8` is `2`. A common use of `%` is to test whether a given integer is even or odd: `N` is even if `N % 2` is zero, and it is odd if `N % 2` is `1`. More generally, you can check whether an integer `N` is evenly divisible by an integer `M` by checking whether `N % M` is zero.

Finally, you might need the ***unary minus*** operator, which takes the negative of a number. For example, `-X` has the same value as `(-1)*X`. For completeness, Java also has a unary plus operator, as in `+X`, even though it doesn't really do anything.

By the way, recall that the `+` operator can also be used to concatenate a value of any type onto a *String*. This is another example of type conversion. In Java, any type can be automatically converted into type *String*.

2.5.2 Increment and Decrement

You'll find that adding `1` to a variable is an extremely common operation in programming. Subtracting `1` from a variable is also pretty common. You might perform the operation of adding `1` to a variable with assignment statements such as:

```
counter  =  counter + 1;
goalsScored  =  goalsScored + 1;
```

The effect of the assignment statement `x = x + 1` is to take the old value of the variable `x`, compute the result of adding `1` to that value, and store the answer as the new value of `x`. The same operation can be accomplished by writing `x++` (or, if you prefer, `++x`). This actually changes the value of `x`, so that it has the same effect as writing "`x = x + 1`". The two statements above could be written

```
counter++;
goalsScored++;
```

Similarly, you could write `x--` (or `--x`) to subtract `1` from `x`. That is, `x--` performs the same computation as `x = x - 1`. Adding `1` to a variable is called ***incrementing*** that variable, and subtracting `1` is called ***decrementing***. The operators `++` and `--` are called the increment operator and the decrement operator, respectively. These operators can be used on variables belonging to any of the numerical types and also on variables of type **char**.

Usually, the operators `++` or `--` are used in statements like "`x++;`" or "`x--;`". These statements are commands to change the value of `x`. However, it is also legal to use `x++`, `++x`, `x--`, or `--x` as expressions, or as parts of larger expressions. That is, you can write things like:

```
y = x++;
y = ++x;
TextIO.putln(--x);
z = (++x) * (y--);
```

The statement "`y = x++;`" has the effects of adding `1` to the value of `x` and, in addition, assigning some value to `y`. The value assigned to `y` is the value of the expression `x++`, which is defined to be the **old** value of `x`, before the `1` is added. Thus, if the value of `x` is `6`, the statement "`y = x++;`" will change the value of `x` to `7`, but it will change the value of `y` to `6` since the value assigned to `y` is the **old** value of `x`. On the other hand, the value of `++x` is defined to be the **new** value of `x`, after the `1` is added. So if `x` is `6`, then the statement "`y = ++x;`" changes the values of both `x` and `y` to `7`. The decrement operator, `--`, works in a similar way.

This can be confusing. My advice is: Don't be confused. Use `++` and `--` only in stand-alone statements, not in expressions. I will follow this advice in almost all examples in these notes.

2.5.3 Relational Operators

Java has boolean variables and boolean-valued expressions that can be used to express conditions that can be either `true` or `false`. One way to form a boolean-valued expression is to compare two values using a ***relational operator***. Relational operators are used to test whether two values are equal, whether one value is greater than another, and so forth. The relational operators in Java are: `==`, `!=`, `<`, `>`, `<=`, and `>=`. The meanings of these operators are:

```
A == B       Is A "equal to" B?
A != B       Is A "not equal to" B?
A < B        Is A "less than" B?
A > B        Is A "greater than" B?
A <= B       Is A "less than or equal to" B?
A >= B       Is A "greater than or equal to" B?
```

These operators can be used to compare values of any of the numeric types. They can also be used to compare values of type **char**. For characters, `<` and `>` are defined according the numeric

Unicode values of the characters. (This might not always be what you want. It is not the same as alphabetical order because all the upper case letters come before all the lower case letters.)

When using boolean expressions, you should remember that as far as the computer is concerned, there is nothing special about boolean values. In the next chapter, you will see how to use them in loop and branch statements. But you can also assign boolean-valued expressions to boolean variables, just as you can assign numeric values to numeric variables.

By the way, the operators `==` and `!=` can be used to compare boolean values. This is occasionally useful. For example, can you figure out what this does:

```
boolean sameSign;
sameSign = ((x > 0) == (y > 0));
```

One thing that you **cannot** do with the relational operators `<`, `>`, `<=`, and `<=` is to use them to compare values of type *String*. You can legally use `==` and `!=` to compare `Strings`, but because of peculiarities in the way objects behave, they might not give the results you want. (The `==` operator checks whether two objects are stored in the same memory location, rather than whether they contain the same value. Occasionally, for some objects, you do want to make such a check—but rarely for strings. I'll get back to this in a later chapter.) Instead, you should use the subroutines `equals()`, `equalsIgnoreCase()`, and `compareTo()`, which were described in Section 2.3, to compare two `Strings`.

2.5.4 Boolean Operators

In English, complicated conditions can be formed using the words "and", "or", and "not." For example, "If there is a test **and** you did **not** study for it...". "And", "or", and "not" are ***boolean operators***, and they exist in Java as well as in English.

In Java, the boolean operator "and" is represented by `&&`. The `&&` operator is used to combine two boolean values. The result is also a boolean value. The result is `true` if **both** of the combined values are `true`, and the result is `false` if **either** of the combined values is `false`. For example, "`(x == 0) && (y == 0)`" is `true` if and only if both `x` is equal to 0 and `y` is equal to 0.

The boolean operator "or" is represented by `||`. (That's supposed to be two of the vertical line characters, `|`.) The expression "`A || B`" is `true` if either `A` is `true` or `B` is `true`, or if both are true. "`A || B`" is `false` only if both `A` and `B` are false.

The operators `&&` and `||` are said to be ***short-circuited*** versions of the boolean operators. This means that the second operand of `&&` or `||` is not necessarily evaluated. Consider the test

```
(x != 0) && (y/x > 1)
```

Suppose that the value of `x` is in fact zero. In that case, the division `y/x` is undefined mathematically. However, the computer will never perform the division, since when the computer evaluates `(x != 0)`, it finds that the result is `false`, and so it knows that (`(x != 0) &&` **anything**) has to be false. Therefore, it doesn't bother to evaluate the second operand, `(y/x > 1)`. The evaluation has been short-circuited and the division by zero is avoided. Without the short-circuiting, there would have been a division by zero. (This may seem like a technicality, and it is. But at times, it will make your programming life a little easier.)

The boolean operator "not" is a unary operator. In Java, it is indicated by `!` and is written in front of its single operand. For example, if `test` is a boolean variable, then

```
test = ! test;
```

will reverse the value of test, changing it from `true` to `false`, or from `false` to `true`.

2.5.5 Conditional Operator

Any good programming language has some nifty little features that aren't really necessary but that let you feel cool when you use them. Java has the conditional operator. It's a ternary operator—that is, it has three operands—and it comes in two pieces, ? and :, that have to be used together. It takes the form

⟨*boolean-expression*⟩ ? ⟨*expression1*⟩ : ⟨*expression2*⟩

The computer tests the value of ⟨***boolean-expression***⟩. If the value is `true`, it evaluates ⟨***expression1***⟩; otherwise, it evaluates ⟨***expression2***⟩. For example:

```
next = (N % 2 == 0) ? (N/2) : (3*N+1);
```

will assign the value `N/2` to `next` if `N` is even (that is, if `N % 2 == 0` is `true`), and it will assign the value `(3*N+1)` to `next` if `N` is odd. (The parentheses in this example are not required, but they do make the expression easier to read.)

2.5.6 Assignment Operators and Type-Casts

You are already familiar with the assignment statement, which uses the symbol "=" to assign the value of an expression to a variable. In fact, = is really an operator in the sense that an assignment can itself be used as an expression or as part of a more complex expression. The value of an assignment such as `A=B` is the same as the value that is assigned to `A`. So, if you want to assign the value of `B` to `A` and test at the same time whether that value is zero, you could say:

```
if ( (A=B) == 0 )...
```

Usually, I would say, **don't do things like that**!

In general, the type of the expression on the right-hand side of an assignment statement must be the same as the type of the variable on the left-hand side. However, in some cases, the computer will automatically convert the value computed by the expression to match the type of the variable. Consider the list of numeric types: **byte**, **short**, **int**, **long**, **float**, **double**. A value of a type that occurs earlier in this list can be converted automatically to a value that occurs later. For example:

```
int A;
double X;
short B;
A = 17;
X = A;    // OK; A is converted to a double
B = A;    // illegal; no automatic conversion
          //       from int to short
```

The idea is that conversion should only be done automatically when it can be done without changing the semantics of the value. Any **int** can be converted to a **double** with the same numeric value. However, there are **int** values that lie outside the legal range of **shorts**. There is simply no way to represent the **int** 100000 as a **short**, for example, since the largest value of type **short** is 32767.

In some cases, you might want to force a conversion that wouldn't be done automatically. For this, you can use what is called a ***type cast***. A type cast is indicated by putting a type name, in parentheses, in front of the value you want to convert. For example,

```
int A;
short B;
A = 17;
B = (short)A;  // OK; A is explicitly type cast
               //      to a value of type short
```

You can do type casts from any numeric type to any other numeric type. However, you should note that you might change the numeric value of a number by type-casting it. For example, `(short)100000` is -31072. (The -31072 is obtained by taking the 4-byte **int** 100000 and throwing away two of those bytes to obtain a **short**—you've lost the real information that was in those two bytes.)

As another example of type casts, consider the problem of getting a random integer between 1 and 6. The function `Math.random()` gives a real number between 0.0 and 0.9999..., and so `6*Math.random()` is between 0.0 and 5.999.... The type-cast operator, `(int)`, can be used to convert this to an integer: `(int)(6*Math.random())`. A real number is cast to an integer by discarding the fractional part. Thus, `(int)(6*Math.random())` is one of the integers 0, 1, 2, 3, 4, and 5. To get a number between 1 and 6, we can add 1: "`(int)(6*Math.random()) + 1`". (The parentheses around `6*Math.random()` are necessary because of precedence rules; without the parentheses, the type cast operator would apply only to the `6`.)

You can also type-cast between the type **char** and the numeric types. The numeric value of a **char** is its Unicode code number. For example, `(char)97` is `'a'`, and `(int)'+'` is `43`. (However, a type conversion from **char** to **int** is automatic and does not have to be indicated with an explicit type cast.)

Java has several variations on the assignment operator, which exist to save typing. For example, "`A += B`" is defined to be the same as "`A = A + B`". Every operator in Java that applies to two operands gives rise to a similar assignment operator. For example:

```
x -= y;     // same as:   x = x - y;
x *= y;     // same as:   x = x * y;
x /= y;     // same as:   x = x / y;
x %= y;     // same as:   x = x % y;   (for integers x and y)
q &&= p;    // same as:   q = q && p;  (for booleans q and p)
```

The combined assignment operator `+=` even works with strings. Recall that when the `+` operator is used with a string as one of the operands, it represents concatenation. Since `str += x` is equivalent to `str = str + x`, when `+=` is used with a string on the left-hand side, it appends the value on the right-hand side onto the string. For example, if `str` has the value "tire", then the statement `str += 'd';` changes the value of `str` to "tired".

2.5.7 Type Conversion of Strings

In addition to automatic type conversions and explicit type casts, there are some other cases where you might want to convert a value of one type into a value of a different type. One common example is the conversion of a *String* value into some other type, such as converting the string `"10"` into the **int** value 10 or the string `"17.42e-2"` into the **double** value 0.1742. In Java, these conversions are handled by built-in functions.

There is a standard class named *Integer* that contains several subroutines and variables related to the **int** data type. (Recall that since **int** is not a class, **int** itself can't contain any subroutines or variables.) In particular, if `str` is any expression of type *String*, then `Integer.parseInt(str)` is a function call that attempts to convert the value of `str` into a

value of type **int**. For example, the value of `Integer.parseInt("10")` is the **int** value 10. If the parameter to `Integer.parseInt` does not represent a legal **int** value, then an error occurs.

Similarly, the standard class named *Double* includes a function `Double.parseDouble` that tries to convert a parameter of type *String* into a value of type **double**. For example, the value of the function call `Double.parseDouble("3.14")` is the **double** value 3.14. (Of course, in practice, the parameter used in `Double.parseDouble` or `Integer.parseInt` would be a variable or expression rather than a constant string.)

Type conversion functions also exist for converting strings into enumerated type values. (Enumerated types, or enums, were introduced in Subsection 2.3.3.) For any enum type, a predefined function named `valueOf` is automatically defined for that type. This is a function that takes a string as parameter and tries to convert it to a value belonging to the enum. The `valueOf` function is part of the enum type, so the name of the enum is part of the full name of the function. For example, if an enum *Suit* is defined as

```
enum Suit { SPADE, DIAMOND, CLUB, HEART }
```

then the name of the type conversion function would be `Suit.valueOf`. The value of the function call `Suit.valueOf("CLUB")` would be the enumerated type value `Suit.CLUB`. For the conversion to succeed, the string must exactly match the simple name of one of the enumerated type constants (**without** the "Suit." in front).

2.5.8 Precedence Rules

If you use several operators in one expression, and if you don't use parentheses to explicitly indicate the order of evaluation, then you have to worry about the precedence rules that determine the order of evaluation. (Advice: don't confuse yourself or the reader of your program; use parentheses liberally.)

Here is a listing of the operators discussed in this section, listed in order from highest precedence (evaluated first) to lowest precedence (evaluated last):

```
Unary operators:              ++, --, !, unary - and +, type-cast
Multiplication and division:  *,  /,  %
Addition and subtraction:     +,  -
Relational operators:         <,  >,  <=,  >=
Equality and inequality:      ==,  !=
Boolean and:                  &&
Boolean or:                   ||
Conditional operator:         ?:
Assignment operators:         =,  +=,  -=,  *=,  /=,  %=
```

Operators on the same line have the same precedence. When operators of the same precedence are strung together in the absence of parentheses, unary operators and assignment operators are evaluated right-to-left, while the remaining operators are evaluated left-to-right. For example, `A*B/C` means `(A*B)/C`, while `A=B=C` means `A=(B=C)`. (Can you see how the expression `A=B=C` might be useful, given that the value of `B=C` as an expression is the same as the value that is assigned to `B`?)

2.6 Programming Environments

Although the Java language is highly standardized, the procedures for creating, compiling, and editing Java programs vary widely from one programming environment to another.

There are two basic approaches: a ***command line environment***, where the user types commands and the computer responds, and an ***integrated development environment*** (IDE), where the user uses the keyboard and mouse to interact with a graphical user interface. While there is just one common command line environment for Java programming, there is a wide variety of IDEs.

I cannot give complete or definitive information on Java programming environments in this section, but I will try to give enough information to let you compile and run the examples from this textbook, at least in a command line environment. There are many IDEs, and I can't cover them all here. I will concentrate on ***Eclipse***, one of the most popular IDEs for Java programming, but some of the information that is presented will apply to other IDEs as well.

One thing to keep in mind is that you do not have to pay any money to do Java programming (aside from buying a computer, of course). Everything that you need can be downloaded for free on the Internet.

2.6.1 Java Development Kit

The basic development system for Java programming is usually referred to as the ***JDK*** (Java Development Kit). It is a part of Java SE, the Java "Standard Edition" (as opposed to Java for servers or for mobile devices). This book requires Java Version 5.0 or higher. Confusingly, the JDKs that are part of Java Versions 5, 6, and 7 are sometimes referred to as JDK 1.5, 1.6, and 1.7. Note that Java SE comes in two versions, a Development Kit version (the JDK) and a Runtime Environment version (the JRE). The Runtime can be used to run Java programs and to view Java applets in Web pages, but it does not allow you to compile your own Java programs. The Development Kit includes the Runtime and adds to it the JDK which lets you compile programs. You need a JDK for use with this textbook.

Java was developed by Sun Microsystems, Inc., which is now a part of the Oracle corporation. Oracle makes the JDK for Windows and Linux available for free download at its Java Web site, http://www.oracle.com/technetwork/java. If you have a Windows computer, it might have come with a Java Runtime, but you might still need to download the JDK. Some versions of Linux come with the JDK either installed by default or on the installation media. If you need to download and install the JDK, be sure to get JDK 5.0 (or higher). As of August 2010, the current version of the JDK is **JDK 6**, and it can be downloaded from

```
http://www.oracle.com/technetwork/java/javase/downloads/index.html
```

Mac OS comes with Java. Recent versions of Mac OS come with Java Version 5 or Version 6, so you will not need to download anything.

If a JDK is properly installed on your computer, you can use the command line environment to compile and run Java programs. Most IDEs also require Java to be installed, so even if you plan to use an IDE for programming, you probably still need a JDK, or at least a JRE.

2.6.2 Command Line Environment

Many modern computer users find the command line environment to be pretty alien and unintuitive. It is certainly very different from the graphical user interfaces that most people are used to. However, it takes only a little practice to learn the basics of the command line environment and to become productive using it.

To use a command line programming environment, you will have to open a window where you can type in commands. In Windows, you can open such a command window by running

the program named ***cmd***. In recent versions of Windows, it can be found in the "Accessories" submenu of the Start menu, under the name "Command Prompt". Alternatively, you can run `cmd` by using the "Run Program" feature in the Start menu, and entering "cmd" as the name of the program. In Mac OS, you want to run the ***Terminal*** program, which can be found in the Utilities folder inside the Applications folder. In Linux, there are several possibilities, including an old program called ***xterm***. In Ubuntu Linux, you can use the "Terminal" command under "Accessories" in the "Applications" menu.

No matter what type of computer you are using, when you open a command window, it will display a prompt of some sort. Type in a command at the prompt and press return. The computer will carry out the command, displaying any output in the command window, and will then redisplay the prompt so that you can type another command. One of the central concepts in the command line environment is the ***current directory*** which contains the files to which commands that you type apply. (The words "directory" and "folder" mean the same thing.) Often, the name of the current directory is part of the command prompt. You can get a list of the files in the current directory by typing in the command ***dir*** (on Windows) or ***ls*** (on Linux and Mac OS). When the window first opens, the current directory is your ***home directory***, where all your files are stored. You can change the current directory using the ***cd*** command with the name of the directory that you want to use. For example, to change into your Desktop directory, type in the command `cd Desktop` and press return.

You should create a directory (that is, a folder) to hold your Java work. For example, create a directory named `javawork` in your home directory. You can do this using your computer's GUI; another way to do it is to open a command window and enter the command `mkdir javawork`. When you want to work on programming, open a command window and enter the command `cd javawork` to change into your work directory. Of course, you can have more than one working directory for your Java work; you can organize your files any way you like.

* * *

The most basic commands for using Java on the command line are ***javac*** and ***java***; `javac` is used to compile Java source code, and `java` is used to run Java stand-alone applications. If a JDK is correctly installed on your computer, it should recognize these commands when you type them in on the command line. Try typing the commands `java -version` and `javac -version` which should tell you which version of Java is installed. If you get a message such as "Command not found," then Java is not correctly installed. If the "java" command works, but "javac" does not, it means that a Java Runtime is installed rather than a Development Kit. (On Windows, after installing the JDK, you need to modify the Windows PATH variable to make this work. See the JDK installation instructions for information about how to do this.)

To test the `javac` command, place a copy of *TextIO.java* into your working directory. (If you downloaded the Web site of this book, you can find it in the directory named `source`; you can use your computer's GUI to copy-and-paste this file into your working directory. Alternatively, you can navigate to *TextIO.java* on the book's Web site and use the "Save As" command in your Web browser to save a copy of the file into your working directory.) Type the command:

```
javac  TextIO.java
```

This will compile `TextIO.java` and will create a bytecode file named `TextIO.class` in the same directory. Note that if the command succeeds, you will not get any response from the computer; it will just redisplay the command prompt to tell you it's ready for another command.

To test the `java` command, copy sample program `Interest2.java` from this book's source directory into your working directory. First, compile the program with the command

```
javac  Interest2.java
```

Remember that for this to succeed, *TextIO* must already be in the same directory. Then you can execute the program using the command

```
java  Interest2
```

Be careful to use **just the name** of the program, *Interest2*, with the `java` command, not the name of the Java source code file or the name of the compiled class file. When you give this command, the program will run. You will be asked to enter some information, and you will respond by typing your answers into the command window, pressing return at the end of the line. When the program ends, you will see the command prompt, and you can enter another command.

You can follow the same procedure to run all of the examples in the early sections of this book. When you start work with applets, you will need a different way to run the applets. That will be discussed later in the book.

* * *

To create your own programs, you will need a ***text editor***. A text editor is a computer program that allows you to create and save documents that contain plain text. It is important that the documents be saved as plain text, that is without any special encoding or formatting information. Word processor documents are not appropriate, unless you can get your word processor to save as plain text. A good text editor can make programming a lot more pleasant. Linux comes with several text editors. On Windows, you can use notepad in a pinch, but you will probably want something better. For Mac OS, you might download the free ***TextWrangler*** application. One possibility that will work on any platform is to use ***jedit***, a good programmer's text editor that is itself written in Java and that can be downloaded for free from www.jedit.org.

To create your own programs, you should open a command line window and `cd` into the working directory where you will store your source code files. Start up your text editor program, such as by double-clicking its icon or selecting it from a Start menu. Type your code into the editor window, or open an existing source code file that you want to modify. Save the file. Remember that the name of a Java source code file must end in ".java", and the rest of the file name must match the name of the class that is defined in the file. Once the file is saved in your working directory, go to the command window and use the `javac` command to compile it, as discussed above. If there are syntax errors in the code, they will be listed in the command window. Each error message contains the line number in the file where the computer found the error. Go back to the editor and try to fix the errors, **save your changes**, and then try the `javac` command again. (It's usually a good idea to just work on the first few errors; sometimes fixing those will make other errors go away.) Remember that when the `javac` command finally succeeds, you will get no message at all. Then you can use the `java` command to run your program, as described above. Once you've compiled the program, you can run it as many times as you like without recompiling it.

That's really all there is to it: Keep both editor and command-line window open. Edit, save, and compile until you have eliminated all the syntax errors. (Always remember to save the file before compiling it—the compiler only sees the saved file, not the version in the editor window.) When you run the program, you might find that it has semantic errors that cause it to run incorrectly. In that case, you have to go back to the edit/save/compile loop to try to find and fix the problem.

2.6.3 IDEs and Eclipse

In an Integrated Development Environment, everything you need to create, compile, and run programs is integrated into a single package, with a graphical user interface that will be familiar to most computer users. There are many different IDEs for Java program development, ranging from fairly simple wrappers around the JDK to highly complex applications with a multitude of features. For a beginning programmer, there is a danger in using an IDE, since the difficulty of learning to use the IDE, on top of the difficulty of learning to program, can be overwhelming. However, for my own programming, I generally use the ***Eclipse*** IDE, and I introduce my students to it after they have had some experience with the command line. Eclipse has a variety of features that are very useful for a beginning programmer. And even though it has many advanced features, its design makes it possible to use Eclipse without understanding its full complexity. Eclipse is used by many professional programmers and is probably the most commonly used Java IDE.

Eclipse is itself written in Java. It requires Java 1.4 or higher to run, and Java 5.0 or higher is recommended. For use with this book, you should be running Eclipse with Java 5.0 or higher. Eclipse requires a Java Runtime Environment, not necessarily a JDK. You should make sure that the JRE or JDK, Version 5.0 or higher is installed on your computer, as described above, **before** you install Eclipse. Eclipse can be downloaded for free from eclipse.org. You can download the "Eclipse IDE for Java Developers."

Another popular choice of IDE is ***Netbeans***, which provides many of the same capabilities as Eclipse. Netbeans can be downloaded from netbeans.org, and Oracle offers downloads of Netbeans on its Java web site. I like Netbeans a little less than Eclipse, and I won't say much about it here. It is, however, quite similar to Eclipse.

The first time you start Eclipse, you will be asked to specify a ***workspace***, which is the directory where all your work will be stored. You can accept the default name, or provide one of your own. When startup is complete, the Eclipse window will be filled by a large "Welcome" screen that includes links to extensive documentation and tutorials. You can close this screen, by clicking the "X" next to the word "Welcome"; you can get back to it later by choosing "Welcome" from the "Help" menu.

The Eclipse GUI consists of one large window that is divided into several sections. Each section contains one or more ***views***. If there are several views in one section, then there will be tabs at the top of the section to select the view that is displayed in that section. Each view displays a different type of information. The whole set of views is called a ***perspective***. Eclipse uses different perspectives, that is different sets of views of different types of information, for different tasks. For compiling and running programs, the only perspective that you will need is the "Java Perspective," which is the default. As you become more experiences, you might want to the use the "Debug Perspective," which has features designed to help you find semantic errors in programs.

The Java Perspective includes a large area in the center of the window where you will create and edit your Java programs. To the left of this is the Package Explorer view, which will contain a list of your Java projects and source code files. To the right are some other views that I don't find very useful, and I suggest that you close them by clicking the small "X" next to the name of each view. Several other views that **will** be useful while you are compiling and running programs appear in a section of the window below the editing area. If you accidently close one of the important views, such as the Package Explorer, you can get it back by selecting it from the "Show View" submenu of the "Window" menu.

* * *

To do any work in Eclipse, you need a ***project***. To start a Java project, go to the "New" submenu in the "File" menu, and select the "Java Project" command. In the window that pops up, it is only necessary to fill in a "Project Name" for the project and click the "Finish" button. The project name can be anything you like. The project should appear in the "Package Explorer" view. Click on the small triangle next to the project name to see the contents of the project. Assuming that you use the default settings, there should be a directory named "src," which is where your Java source code files will go. It also contains the "JRE System Library"; this is the collection of standard built-in classes that come with Java.

To run the *TextIO* based examples from this textbook, you must add the source code file *TextIO.java* to your project. If you have downloaded the Web site of this book, you can find a copy of *TextIO.java* in the source directory. Alternatively, you can navigate to the file online and use the "Save As" command of your Web browser to save a copy of the file onto your computer. The easiest way to get *TextIO* into your project is to locate the source code file on your computer and drag the file icon onto the project name in the Eclipse window. If that doesn't work, you can try using copy-and-paste: Right-click the file icon (or control-click on Mac OS), select "Copy" from the pop-up menu, right-click the project name in the Eclipse window, and select "Paste". If you also have trouble with that, you can try using the "Import" command in Eclipse's "File" menu; select "File System" (under "General") in the window that pops up, click "Next", and provide the necessary information in the next window. (Unfortunately, using the file import window is rather complicated. If you find that you have to use it, you should consult the Eclipse documentation about it.) In any case, *TextIO* should appear in the src directory of your project, inside a ***package*** named "default package". Once a file is in this list, you can open it by double-clicking it; it will appear in the editing area of the Eclipse window.

To run any of the Java programs from this textbook, copy the source code file into your Eclipse Java project in the same way that you did for TextIO.java. To run the program, right-click the file name in the Package Explorer view (or control-click in Mac OS). In the menu that pops up, go to the "Run As" submenu, and select "Java Application". The program will be executed. If the program writes to standard output, the output will appear in the "Console" view, in the area of the Eclipse window below the editing area. If the program uses *TextIO* for input, you will have to type the required input into the "Console" view—**click the "Console" view before you start typing**, so that the characters that you type will be sent to the correct part of the window. (Note that if you don't like doing I/O in the "Console" view, you can use an alternative version of *TextIO.java* that opens a separate window for I/O. You can find this "GUI" version of TextIO in a directory named `TextIO-GUI` inside this textbook's source directory.)

You can have more than one program in the same Eclipse project, or you can create additional projects to organize your work better. Remember to place a copy of *TextIO.java* in any project that requires it.

* * *

To create your own Java program, you must create a new Java class. To do this, right-click the Java project name in the "Project Explorer" view. Go to the "New" submenu of the popup menu, and select "Class". (Alternatively, there is a small icon at the top of the Eclipse window that you can click to create a new Java class.) In the window that opens, type in the name of the class, and click the "Finish" button. The class name must be a legal Java identifier. Note that you want the name of the class, not the name of the source code file, so don't add ".java" at the end of the name. The class should appear inside the "default package," and it should

automatically open in the editing area so that you can start typing in your program.

Eclipse has several features that aid you as you type your code. It will underline any syntax error with a jagged red line, and in some cases will place an error marker in the left border of the edit window. If you hover the mouse cursor over the error marker or over the error itself, a description of the error will appear. Note that you do not have to get rid of every error immediately as you type; some errors will go away as you type in more of the program. If an error marker displays a small "light bulb," Eclipse is offering to try to fix the error for you. Click the light bulb to get a list of possible fixes, then double click the fix that you want to apply. For example, if you use an undeclared variable in your program, Eclipse will offer to declare it for you. You can actually use this error-correcting feature to get Eclipse to write certain types of code for you! Unfortunately, you'll find that you won't understand a lot of the proposed fixes until you learn more about the Java language, and it is **not** a good idea to apply a fix that you don't understand—often that will just make things worse in the end.

Eclipse will also look for spelling errors in comments and will underline them with jagged red lines. Hover your mouse over the error to get a list of possible correct spellings.

Another essential Eclipse feature is ***content assist***. Content assist can be invoked by typing Control-Space. It will offer possible completions of whatever you are typing at the moment. For example, if you type part of an identifier and hit Control-Space, you will get a list of identifiers that start with the characters that you have typed; use the up and down arrow keys to select one of the items in the list, and press Return or Enter. (Or hit Escape to dismiss the list.) If there is only one possible completion when you hit Control-Space, it will be inserted automatically. By default, Content Assist will also pop up automatically, after a short delay, when you type a period or certain other characters. For example, if you type "`TextIO.`" and pause for just a fraction of a second, you will get a list of all the subroutines in the *TextIO* class. Personally, I find this auto-activation annoying. You can disable it in the Eclipse Preferences. (Look under Java / Editor / Content Assist, and turn off the "Enable auto activation" option.) You can still call up Code Assist manually with Control-Space.

Once you have an error-free program, you can run it as described above, by right-clicking its name in the Package Explorer and using "Run As / Java Application". You can also right-click on the program itself in an editor window. If you find a problem when you run it, it's very easy to go back to the editor, make changes, and run it again. Note that using Eclipse, there is no explicit "compile" command. The source code files in your project are automatically compiled, and are re-compiled whenever you modify them.

If you use Netbeans instead of Eclipse, the procedures are similar. You still have to create new project (of type "Java Application"). You can add an existing source code file to a project by dragging the file onto the "Source Packages" folder in the project, and you can create your own classes by right-clicking the project name and selecting New/Java Class. To run a program, right-click the file that contains the main routine, and select the "Run File" command. Netbeans has a "Code Completion" feature that is similar to Eclipse's "Content Assist." One thing that you have to watch with Netbeans is that it might want to create classes in (non-default) packages; when you create a New Java Class, make sure that the "Package" input box is left blank.

2.6.4 The Problem of Packages

Every class in Java is contained in something called a ***package***. Classes that are not explicitly put into a different package are in the "default" package. Almost all the examples in this

textbook are in the default package, and I will not even discuss packages in any depth until Section 4.5. However, some IDEs might force you to pay attention to packages.

When you create a class in Eclipse, you might notice a message that says that "The use of the default package is discouraged." Although this is true, I have chosen to use it anyway, since it seems easier for beginning programmers to avoid the whole issue of packages, at least at first. Some IDEs, like Netbeans, are even less willing than Eclipse to use the default package: Netbeans inserts a package name automatically in the class creation dialog, and you have to delete that name if you want to create the class in the default package. If you do create a class in a package, the source code starts with a line that specifies which package the class is in. For example, if the class is in a package named `test.pkg`, then the first line of the source code will be

```
package test.pkg;
```

In an IDE, this will not cause any problem unless the program you are writing depends on *TextIO*. You will not be able to use *TextIO* in a program unless *TextIO* is in the same package as the program. You can put TextIO in a named, non-default package, but you have to modify the source code file *TextIO.java* to specify the package: Just add a `package` statement like the one shown above to the very beginning of the file, with the appropriate package name. (The IDE might do this for you, if you copy TextIO.java into a non-default package.) Once you've done this, the example should run in the same way as if it were in the default package.

By the way, if you use packages in a command-line environment, other complications arise. For example, if a class is in a package named `test.pkg`, then the source code file must be in a subdirectory named "pkg" inside a directory named "test" that is in turn inside your main Java working directory. Nevertheless, when you compile or execute the program, you should be in the main directory, not in a subdirectory. When you compile the source code file, you have to include the name of the directory in the command: Use "`javac test/pkg/ClassName.java`" on Linux or Mac OS, or "`javac test\pkg\ClassName.java`" on Windows. The command for executing the program is then "`java test.pkg.ClassName`", with a period separating the package name from the class name. However, you will not need to worry about any of that when working with almost all of the examples in this book.

Exercises for Chapter 2

1. Write a program that will print your initials to standard output in letters that are nine lines tall. Each big letter should be made up of a bunch of *'s. For example, if your initials were "DJE", then the output would look something like:

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

2. Write a program that simulates rolling a pair of dice. You can simulate rolling one die by choosing one of the integers 1, 2, 3, 4, 5, or 6 at random. The number you pick represents the number on the die after it is rolled. As pointed out in Section 2.5, The expression

```
(int)(Math.random()*6) + 1
```

does the computation you need to select a random integer between 1 and 6. You can assign this value to a variable to represent one of the dice that are being rolled. Do this twice and add the results together to get the total roll. Your program should report the number showing on each die as well as the total roll. For example:

```
The first die comes up 3
The second die comes up 5
Your total roll is 8
```

3. Write a program that asks the user's name, and then greets the user by name. Before outputting the user's name, convert it to upper case letters. For example, if the user's name is Fred, then the program should respond "Hello, FRED, nice to meet you!".

4. Write a program that helps the user count his change. The program should ask how many quarters the user has, then how many dimes, then how many nickels, then how many pennies. Then the program should tell the user how much money he has, expressed in dollars.

5. If you have `N` eggs, then you have `N/12` dozen eggs, with `N%12` eggs left over. (This is essentially the definition of the `/` and `%` operators for integers.) Write a program that asks the user how many eggs she has and then tells the user how many dozen eggs she has and how many extra eggs are left over.

A gross of eggs is equal to 144 eggs. Extend your program so that it will tell the user how many gross, how many dozen, and how many left over eggs she has. For example, if the user says that she has 1342 eggs, then your program would respond with

```
Your number of eggs is 9 gross, 3 dozen, and 10
```

since 1342 is equal to 9*144 + 3*12 + 10.

6. Suppose that a file named "testdata.txt" contains the following information: The first line of the file is the name of a student. Each of the next three lines contains an integer. The integers are the student's scores on three exams. Write a program that will read the information in the file and display (on standard output) a message the contains the name of the student and the student's average grade on the three exams. The average is obtained by adding up the individual exam grades and then dividing by the number of exams.

Quiz on Chapter 2

1. Briefly explain what is meant by the *syntax* and the *semantics* of a programming language. Give an example to illustrate the difference between a syntax error and a semantics error.

2. What does the computer do when it executes a variable declaration statement. Give an example.

3. What is a *type,* as this term relates to programming?

4. One of the primitive types in Java is *boolean.* What is the **boolean** type? Where are boolean values used? What are its possible values?

5. Give the meaning of each of the following Java operators:

a) `++`

b) `&&`

c) `!=`

6. Explain what is meant by an *assignment statement,* and give an example. What are assignment statements used for?

7. What is meant by *precedence* of operators?

8. What is a *literal*?

9. In Java, classes have two fundamentally different purposes. What are they?

10. What is the difference between the statement "`x = TextIO.getDouble();`" and the statement "`x = TextIO.getlnDouble();`"

11. Explain why the value of the expression `2 + 3 + "test"` is the string `"5test"` while the value of the expression `"test" + 2 + 3` is the string `"test23"`. What is the value of `"test" + 2 * 3` ?

12. Integrated Development Environments such as Eclipse often use ***syntax coloring***, which assigns various colors to the characters in a program to reflect the syntax of the language. A student notices that Eclipse colors the word *String* differently from **int**, **double**, and **boolean**. The student asks why *String* should be a different color, since all these words are names of types. What's the answer to the student's question?

Chapter 3

Programming in the Small II: Control

THE BASIC BUILDING BLOCKS of programs—variables, expressions, assignment statements, and subroutine call statements—were covered in the previous chapter. Starting with this chapter, we look at how these building blocks can be put together to build complex programs with more interesting behavior.

Since we are still working on the level of "programming in the small" in this chapter, we are interested in the kind of complexity that can occur within a single subroutine. On this level, complexity is provided by ***control structures***. The two types of control structures, loops and branches, can be used to repeat a sequence of statements over and over or to choose among two or more possible courses of action. Java includes several control structures of each type, and we will look at each of them in some detail.

This chapter will also begin the study of program design. Given a problem, how can you come up with a program to solve that problem? We'll look at a partial answer to this question in Section 3.2.

3.1 Blocks, Loops, and Branches

THE ABILITY OF A COMPUTER TO PERFORM complex tasks is built on just a few ways of combining simple commands into control structures. In Java, there are just six such structures that are used to determine the normal flow of control in a program—and, in fact, just three of them would be enough to write programs to perform any task. The six control structures are: the ***block***, the ***while loop***, the ***do..while loop***, the ***for loop***, the ***if statement***, and the ***switch statement***. Each of these structures is considered to be a single "statement," but each is in fact a **structured** statement that can contain one or more other statements inside itself.

3.1.1 Blocks

The ***block*** is the simplest type of structured statement. Its purpose is simply to group a sequence of statements into a single statement. The format of a block is:

```
{
    ⟨statements⟩
}
```

That is, it consists of a sequence of statements enclosed between a pair of braces, "`{`" and "`}`". In fact, it is possible for a block to contain no statements at all; such a block is called an ***empty block***, and can actually be useful at times. An empty block consists of nothing but an empty pair of braces. Block statements usually occur inside other statements, where their purpose is to group together several statements into a unit. However, a block can be legally used wherever a statement can occur. There is one place where a block is required: As you might have already noticed in the case of the `main` subroutine of a program, the definition of a subroutine is a block, since it is a sequence of statements enclosed inside a pair of braces.

I should probably note again at this point that Java is what is called a free-format language. There are no syntax rules about how the language has to be arranged on a page. So, for example, you could write an entire block on one line if you want. But as a matter of good programming style, you should lay out your program on the page in a way that will make its structure as clear as possible. In general, this means putting one statement per line and using indentation to indicate statements that are contained inside control structures. This is the format that I will generally use in my examples.

Here are two examples of blocks:

```
{
   System.out.print("The answer is ");
   System.out.println(ans);
}

{  // This block exchanges the values of x and y
   int temp;      // A temporary variable for use in this block.
   temp = x;      // Save a copy of the value of x in temp.
   x = y;         // Copy the value of y into x.
   y = temp;      // Copy the value of temp into y.
}
```

In the second example, a variable, `temp`, is declared inside the block. This is perfectly legal, and it is good style to declare a variable inside a block if that variable is used nowhere else but inside the block. A variable declared inside a block is completely inaccessible and invisible from outside that block. When the computer executes the variable declaration statement, it allocates memory to hold the value of the variable. When the block ends, that memory is discarded (that is, made available for reuse). The variable is said to be ***local*** to the block. There is a general concept called the "scope" of an identifier. The ***scope*** of an identifier is the part of the program in which that identifier is valid. The scope of a variable defined inside a block is limited to that block, and more specifically to the part of the block that comes after the declaration of the variable.

3.1.2 The Basic While Loop

The block statement by itself really doesn't affect the flow of control in a program. The five remaining control structures do. They can be divided into two classes: loop statements and branching statements. You really just need one control structure from each category in order to have a completely general-purpose programming language. More than that is just convenience. In this section, I'll introduce the `while` loop and the `if` statement. I'll give the full details of these statements and of the other three control structures in later sections.

A ***while loop*** is used to repeat a given statement over and over. Of course, it's not likely that you would want to keep repeating it forever. That would be an ***infinite loop***, which is

generally a bad thing. (There is an old story about computer pioneer Grace Murray Hopper, who read instructions on a bottle of shampoo telling her to "lather, rinse, repeat." As the story goes, she claims that she tried to follow the directions, but she ran out of shampoo. (In case you don't get it, this is a joke about the way that computers mindlessly follow instructions.))

To be more specific, a `while` loop will repeat a statement over and over, but only so long as a specified condition remains true. A `while` loop has the form:

```
while (⟨boolean-expression⟩)
     ⟨statement⟩
```

Since the statement can be, and usually is, a block, many `while` loops have the form:

```
while (⟨boolean-expression⟩) {
    ⟨statements⟩
}
```

Some programmers think that the braces should always be included as a matter of style, even when there is only one statement between them, but I don't always follow that advice myself.

The semantics of the `while` statement go like this: When the computer comes to a `while` statement, it evaluates the ⟨*boolean-expression*⟩, which yields either `true` or `false` as its value. If the value is `false`, the computer skips over the rest of the `while` loop and proceeds to the next command in the program. If the value of the expression is `true`, the computer executes the ⟨*statement*⟩ or block of ⟨*statements*⟩ inside the loop. Then it returns to the beginning of the `while` loop and repeats the process. That is, it re-evaluates the ⟨*boolean-expression*⟩, ends the loop if the value is `false`, and continues it if the value is `true`. This will continue over and over until the value of the expression is `false`; if that never happens, then there will be an infinite loop.

Here is an example of a `while` loop that simply prints out the numbers 1, 2, 3, 4, 5:

```
int number;   // The number to be printed.
number = 1;   // Start with 1.
while ( number < 6 ) {  // Keep going as long as number is < 6.
    System.out.println(number);
    number = number + 1;  // Go on to the next number.
}
System.out.println("Done!");
```

The variable `number` is initialized with the value 1. So the first time through the `while` loop, when the computer evaluates the expression "`number < 6`", it is asking whether 1 is less than 6, which is `true`. The computer therefore proceeds to execute the two statements inside the loop. The first statement prints out "1". The second statement adds 1 to `number` and stores the result back into the variable `number`; the value of `number` has been changed to 2. The computer has reached the end of the loop, so it returns to the beginning and asks again whether `number` is less than 6. Once again this is true, so the computer executes the loop again, this time printing out 2 as the value of `number` and then changing the value of `number` to 3. It continues in this way until eventually `number` becomes equal to 6. At that point, the expression "`number < 6`" evaluates to `false`. So, the computer jumps past the end of the loop to the next statement and prints out the message "Done!". Note that when the loop ends, the value of `number` is 6, but the last value that was printed was 5.

By the way, you should remember that you'll never see a `while` loop standing by itself in a real program. It will always be inside a subroutine which is itself defined inside some class. As an example of a `while` loop used inside a complete program, here is a little program

that computes the interest on an investment over several years. This is an improvement over examples from the previous chapter that just reported the results for one year:

```
/**
 *  This class implements a simple program that will compute the amount of
 *  interest that is earned on an investment over a period of 5 years.  The
 *  initial amount of the investment and the interest rate are input by the
 *  user.  The value of the investment at the end of each year is output.
 */

public class Interest3 {


   public static void main(String[] args) {

      double principal;  // The value of the investment.
      double rate;       // The annual interest rate.

      /* Get the initial investment and interest rate from the user. */

      System.out.print("Enter the initial investment: ");
      principal = TextIO.getlnDouble();

      System.out.println();
      System.out.println("Enter the annual interest rate.");
      System.out.print("Enter a decimal, not a percentage: ");
      rate = TextIO.getlnDouble();
      System.out.println();

      /* Simulate the investment for 5 years. */

      int years;  // Counts the number of years that have passed.

      years = 0;
      while (years < 5) {
         double interest;  // Interest for this year.
         interest = principal * rate;
         principal = principal + interest;     // Add it to principal.
         years = years + 1;    // Count the current year.
         System.out.print("The value of the investment after ");
         System.out.print(years);
         System.out.print(" years is $");
         System.out.printf("%1.2f", principal);
         System.out.println();
      } // end of while loop

   } // end of main()

} // end of class Interest3
```

You should study this program, and make sure that you understand what the computer does step-by-step as it executes the `while` loop.

3.1.3 The Basic If Statement

An ***if statement*** tells the computer to take one of two alternative courses of action, depending on whether the value of a given boolean-valued expression is true or false. It is an example of a "branching" or "decision" statement. An `if` statement has the form:

```
if ( ⟨boolean-expression⟩ )
    ⟨statement⟩
else
    ⟨statement⟩
```

When the computer executes an `if` statement, it evaluates the boolean expression. If the value is `true`, the computer executes the first statement and skips the statement that follows the "`else`". If the value of the expression is `false`, then the computer skips the first statement and executes the second one. Note that in any case, one and only one of the two statements inside the `if` statement is executed. The two statements represent alternative courses of action; the computer decides between these courses of action based on the value of the boolean expression.

In many cases, you want the computer to choose between doing something and not doing it. You can do this with an `if` statement that omits the `else` part:

```
if ( ⟨boolean-expression⟩ )
    ⟨statement⟩
```

To execute this statement, the computer evaluates the expression. If the value is `true`, the computer executes the ⟨*statement*⟩ that is contained inside the `if` statement; if the value is `false`, the computer skips over that ⟨*statement*⟩.

Of course, either or both of the ⟨*statements*⟩ in an `if` statement can be a block, and again many programmers prefer to add the braces even when they contain just a single statement. So an `if` statement often looks like:

```
if ( ⟨boolean-expression⟩ ) {
    ⟨statements⟩
}
else {
    ⟨statements⟩
}
```

or:

```
if ( ⟨boolean-expression⟩ ) {
    ⟨statements⟩
}
```

As an example, here is an `if` statement that exchanges the value of two variables, `x` and `y`, but only if `x` is greater than `y` to begin with. After this `if` statement has been executed, we can be sure that the value of `x` is definitely less than or equal to the value of `y`:

```
if ( x > y ) {
    int temp;      // A temporary variable for use in this block.
    temp = x;      // Save a copy of the value of x in temp.
    x = y;         // Copy the value of y into x.
    y = temp;      // Copy the value of temp into y.
}
```

Finally, here is an example of an `if` statement that includes an `else` part. See if you can figure out what it does, and why it would be used:

```
if ( years > 1 ) {  // handle case for 2 or more years
    System.out.print("The value of the investment after ");
    System.out.print(years);
    System.out.print(" years is $");
}
```

```
else {  // handle case for 1 year
    System.out.print("The value of the investment after 1 year is $");
}  // end of if statement
System.out.printf("%1.2f", principal);  // this is done in any case
```

I'll have more to say about control structures later in this chapter. But you already know the essentials. If you never learned anything more about control structures, you would already know enough to perform any possible computing task. Simple looping and branching are all you really need!

3.2 Algorithm Development

PROGRAMMING IS DIFFICULT (like many activities that are useful and worthwhile—and like most of those activities, it can also be rewarding and a lot of fun). When you write a program, you have to tell the computer every small detail of what to do. And you have to get everything exactly right, since the computer will blindly follow your program exactly as written. How, then, do people write any but the most simple programs? It's not a big mystery, actually. It's a matter of learning to think in the right way.

A program is an expression of an idea. A programmer starts with a general idea of a task for the computer to perform. Presumably, the programmer has some idea of how to perform the task by hand, at least in general outline. The problem is to flesh out that outline into a complete, unambiguous, step-by-step procedure for carrying out the task. Such a procedure is called an "algorithm." (Technically, an ***algorithm*** is an unambiguous, step-by-step procedure that terminates after a finite number of steps; we don't want to count procedures that go on forever.) An algorithm is not the same as a program. A program is written in some particular programming language. An algorithm is more like the **idea** behind the program, but it's the idea of the **steps** the program will take to perform its task, not just the idea of the **task** itself. When describing an algorithm, the steps don't necessarily have to be specified in complete detail, as long as the steps are unambiguous and it's clear that carrying out the steps will accomplish the assigned task. An algorithm can be expressed in any language, including English. Of course, an algorithm can only be expressed as a program if all the details have been filled in.

So, where do algorithms come from? Usually, they have to be developed, often with a lot of thought and hard work. Skill at algorithm development is something that comes with practice, but there are techniques and guidelines that can help. I'll talk here about some techniques and guidelines that are relevant to "programming in the small," and I will return to the subject several times in later chapters.

3.2.1 Pseudocode and Stepwise Refinement

When programming in the small, you have a few basics to work with: variables, assignment statements, and input/output routines. You might also have some subroutines, objects, or other building blocks that have already been written by you or someone else. (Input/output routines fall into this class.) You can build sequences of these basic instructions, and you can also combine them into more complex control structures such as `while` loops and `if` statements.

Suppose you have a task in mind that you want the computer to perform. One way to proceed is to write a description of the task, and take that description as an outline of the algorithm you want to develop. Then you can refine and elaborate that description, gradually adding steps and detail, until you have a complete algorithm that can be translated directly

into programming language. This method is called ***stepwise refinement***, and it is a type of top-down design. As you proceed through the stages of stepwise refinement, you can write out descriptions of your algorithm in ***pseudocode***—informal instructions that imitate the structure of programming languages without the complete detail and perfect syntax of actual program code.

As an example, let's see how one might develop the program from the previous section, which computes the value of an investment over five years. The task that you want the program to perform is: "Compute and display the value of an investment for each of the next five years, where the initial investment and interest rate are to be specified by the user." You might then write—or at least think—that this can be expanded as:

```
Get the user's input
Compute the value of the investment after 1 year
Display the value
Compute the value after 2 years
Display the value
Compute the value after 3 years
Display the value
Compute the value after 4 years
Display the value
Compute the value after 5 years
Display the value
```

This is correct, but rather repetitive. And seeing that repetition, you might notice an opportunity to use a loop. A loop would take less typing. More important, it would be more **general**: Essentially the same loop will work no matter how many years you want to process. So, you might rewrite the above sequence of steps as:

```
Get the user's input
while there are more years to process:
    Compute the value after the next year
    Display the value
```

Following this algorithm would certainly solve the problem, but for a computer we'll have to be more explicit about how to "Get the user's input," how to "Compute the value after the next year," and what it means to say "there are more years to process." We can expand the step, "Get the user's input" into

```
Ask the user for the initial investment
Read the user's response
Ask the user for the interest rate
Read the user's response
```

To fill in the details of the step "Compute the value after the next year," you have to know how to do the computation yourself. (Maybe you need to ask your boss or professor for clarification?) Let's say you know that the value is computed by adding some interest to the previous value. Then we can refine the `while` loop to:

```
while there are more years to process:
    Compute the interest
    Add the interest to the value
    Display the value
```

As for testing whether there are more years to process, the only way that we can do that is by counting the years ourselves. This displays a very common pattern, and you should expect to use something similar in a lot of programs: We have to start with zero years, add one each time we process a year, and stop when we reach the desired number of years. So the `while` loop becomes:

```
years = 0
while years < 5:
    years = years + 1
    Compute the interest
    Add the interest to the value
    Display the value
```

We still have to know how to compute the interest. Let's say that the interest is to be computed by multiplying the interest rate by the current value of the investment. Putting this together with the part of the algorithm that gets the user's inputs, we have the complete algorithm:

```
Ask the user for the initial investment
Read the user's response
Ask the user for the interest rate
Read the user's response
years = 0
while years < 5:
    years = years + 1
    Compute interest = value * interest rate
    Add the interest to the value
    Display the value
```

Finally, we are at the point where we can translate pretty directly into proper programming-language syntax. We still have to choose names for the variables, decide exactly what we want to say to the user, and so forth. Having done this, we could express our algorithm in Java as:

```
double principal, rate, interest;  // declare the variables
int years;
System.out.print("Type initial investment: ");
principal = TextIO.getlnDouble();
System.out.print("Type interest rate: ");
rate = TextIO.getlnDouble();
years = 0;
while (years < 5) {
   years = years + 1;
   interest = principal * rate;
   principal = principal + interest;
   System.out.println(principal);
}
```

This still needs to be wrapped inside a complete program, it still needs to be commented, and it really needs to print out more information in a nicer format for the user. But it's essentially the same program as the one in the previous section. (Note that the pseudocode algorithm uses indentation to show which statements are inside the loop. In Java, indentation is completely ignored by the computer, so you need a pair of braces to tell the computer which statements are in the loop. If you leave out the braces, the only statement inside the loop would be "`years = years + 1;`". The other statements would only be executed once, after the loop

ends. The nasty thing is that the computer won't notice this error for you, like it would if you left out the parentheses around "`(years < 5)`". The parentheses are required by the syntax of the `while` statement. The braces are only required semantically. The computer can recognize syntax errors but not semantic errors.)

One thing you should have noticed here is that my original specification of the problem—"Compute and display the value of an investment for each of the next five years"—was far from being complete. Before you start writing a program, you should make sure you have a complete specification of exactly what the program is supposed to do. In particular, you need to know what information the program is going to input and output and what computation it is going to perform. Here is what a reasonably complete specification of the problem might look like in this example:

> "Write a program that will compute and display the value of an investment for each of the next five years. Each year, interest is added to the value. The interest is computed by multiplying the current value by a fixed interest rate. Assume that the initial value and the rate of interest are to be input by the user when the program is run."

3.2.2 The 3N+1 Problem

Let's do another example, working this time with a program that you haven't already seen. The assignment here is an abstract mathematical problem that is one of my favorite programming exercises. This time, we'll start with a more complete specification of the task to be performed:

> "Given a positive integer, N, define the '3N+1' sequence starting from N as follows: If N is an even number, then divide N by two; but if N is odd, then multiply N by 3 and add 1. Continue to generate numbers in this way until N becomes equal to 1. For example, starting from N = 3, which is odd, we multiply by 3 and add 1, giving N = 3*3+1 = 10. Then, since N is even, we divide by 2, giving N = 10/2 = 5. We continue in this way, stopping when we reach 1, giving the complete sequence: 3, 10, 5, 16, 8, 4, 2, 1.
>
> "Write a program that will read a positive integer from the user and will print out the 3N+1 sequence starting from that integer. The program should also count and print out the number of terms in the sequence."

A general outline of the algorithm for the program we want is:

```
Get a positive integer N from the user.
Compute, print, and count each number in the sequence.
Output the number of terms.
```

The bulk of the program is in the second step. We'll need a loop, since we want to keep computing numbers until we get 1. To put this in terms appropriate for a `while` loop, we need to know when to **continue** the loop rather than when to stop it: We want to continue as long as the number is **not** 1. So, we can expand our pseudocode algorithm to:

```
Get a positive integer N from the user;
while N is not 1:
    Compute N = next term;
    Output N;
    Count this term;
Output the number of terms;
```

In order to compute the next term, the computer must take different actions depending on whether N is even or odd. We need an `if` statement to decide between the two cases:

```
Get a positive integer N from the user;
while N is not 1:
    if N is even:
       Compute N = N/2;
    else
       Compute N = 3 * N + 1;
    Output N;
    Count this term;
Output the number of terms;
```

We are almost there. The one problem that remains is counting. Counting means that you start with zero, and every time you have something to count, you add one. We need a variable to do the counting. (Again, this is a common pattern that you should expect to see over and over.) With the counter added, we get:

```
Get a positive integer N from the user;
Let counter = 0;
while N is not 1:
    if N is even:
       Compute N = N/2;
    else
       Compute N = 3 * N + 1;
    Output N;
    Add 1 to counter;
Output the counter;
```

We still have to worry about the very first step. How can we get a **positive** integer from the user? If we just read in a number, it's possible that the user might type in a negative number or zero. If you follow what happens when the value of N is negative or zero, you'll see that the program will go on forever, since the value of N will never become equal to 1. This is bad. In this case, the problem is probably no big deal, but in general you should try to write programs that are foolproof. One way to fix this is to keep reading in numbers until the user types in a positive number:

```
Ask user to input a positive number;
Let N be the user's response;
while N is not positive:
   Print an error message;
   Read another value for N;
Let counter = 0;
while N is not 1:
    if N is even:
       Compute N = N/2;
    else
       Compute N = 3 * N + 1;
```

```
      Output N;
      Add 1 to counter;
   Output the counter;
```

The first `while` loop will end only when N is a positive number, as required. (A common beginning programmer's error is to use an `if` statement instead of a `while` statement here: "If N is not positive, ask the user to input another value." The problem arises if the second number input by the user is also non-positive. The `if` statement is only executed once, so the second input number is never tested, and the program proceeds into an infinite loop. With the `while` loop, after the second number is input, the computer jumps back to the beginning of the loop and tests whether the second number is positive. If not, it asks the user for a third number, and it will continue asking for numbers until the user enters an acceptable input.)

Here is a Java program implementing this algorithm. It uses the operators `<=` to mean "is less than or equal to" and `!=` to mean "is not equal to." To test whether N is even, it uses "`N % 2 == 0`". All the operators used here were discussed in Section 2.5.

```
/**
 * This program prints out a 3N+1 sequence starting from a positive
 * integer specified by the user.  It also counts the number of
 * terms in the sequence, and prints out that number.
 */
public class ThreeN1 {

     public static void main(String[] args) {

        int N;       // for computing terms in the sequence
        int counter; // for counting the terms

        TextIO.put("Starting point for sequence: ");
        N = TextIO.getlnInt();
        while (N <= 0) {
           TextIO.put("The starting point must be positive. Please try again: ");
           N = TextIO.getlnInt();
        }
        // At this point, we know that N > 0

        counter = 0;
        while (N != 1) {
            if (N % 2 == 0)
               N = N / 2;
            else
               N = 3 * N + 1;
            TextIO.putln(N);
            counter = counter + 1;
        }

        TextIO.putln();
        TextIO.put("There were ");
        TextIO.put(counter);
        TextIO.putln(" terms in the sequence.");

    }  // end of main()

}  // end of class ThreeN1
```

Two final notes on this program: First, you might have noticed that the first term of the sequence—the value of N input by the user—is not printed or counted by this program. Is this an error? It's hard to say. Was the specification of the program careful enough to decide? This is the type of thing that might send you back to the boss/professor for clarification. The problem (if it is one!) can be fixed easily enough. Just replace the line "counter = 0" before the while loop with the two lines:

```
TextIO.putln(N);   // print out initial term
counter = 1;       // and count it
```

Second, there is the question of why this problem is at all interesting. Well, it's interesting to mathematicians and computer scientists because of a simple question about the problem that they haven't been able to answer: Will the process of computing the 3N+1 sequence finish after a finite number of steps for all possible starting values of N? Although individual sequences are easy to compute, no one has been able to answer the general question. To put this another way, no one knows whether the process of computing 3N+1 sequences can properly be called an algorithm, since an algorithm is required to terminate after a finite number of steps! (This discussion assumes that the value of `N` can take on arbitrarily large integer values, which is not true for a variable of type **int** in a Java program. When the value of `N` in the program becomes too large to be represented as a 32-bit **int**, the values output by the program are no longer mathematically correct. See Exercise 8.2)

3.2.3 Coding, Testing, Debugging

It would be nice if, having developed an algorithm for your program, you could relax, press a button, and get a perfectly working program. Unfortunately, the process of turning an algorithm into Java source code doesn't always go smoothly. And when you do get to the stage of a working program, it's often only working in the sense that it does **something**. Unfortunately not what you want it to do.

After program design comes coding: translating the design into a program written in Java or some other language. Usually, no matter how careful you are, a few syntax errors will creep in from somewhere, and the Java compiler will reject your program with some kind of error message. Unfortunately, while a compiler will always detect syntax errors, it's not very good about telling you exactly what's wrong. Sometimes, it's not even good about telling you where the real error is. A spelling error or missing "`{`" on line 45 might cause the compiler to choke on line 105. You can avoid lots of errors by making sure that you really understand the syntax rules of the language and by following some basic programming guidelines. For example, I never type a "`{`" without typing the matching "`}`". Then I go back and fill in the statements between the braces. A missing or extra brace can be one of the hardest errors to find in a large program. Always, always indent your program nicely. If you change the program, change the indentation to match. It's worth the trouble. Use a consistent naming scheme, so you don't have to struggle to remember whether you called that variable `interestrate` or `interestRate`. In general, when the compiler gives multiple error messages, don't try to fix the second error message from the compiler until you've fixed the first one. Once the compiler hits an error in your program, it can get confused, and the rest of the error messages might just be guesses. Maybe the best advice is: Take the time to understand the error before you try to fix it. Programming is not an experimental science.

When your program compiles without error, you are still not done. You have to test the program to make sure it works correctly. Remember that the goal is not to get the right output

for the two sample inputs that the professor gave in class. The goal is a program that will work correctly for all reasonable inputs. Ideally, when faced with an unreasonable input, it should respond by gently chiding the user rather than by crashing. Test your program on a wide variety of inputs. Try to find a set of inputs that will test the full range of functionality that you've coded into your program. As you begin writing larger programs, write them in stages and test each stage along the way. You might even have to write some extra code to do the testing—for example to call a subroutine that you've just written. You don't want to be faced, if you can avoid it, with 500 newly written lines of code that have an error in there *somewhere.*

The point of testing is to find ***bugs***—semantic errors that show up as incorrect behavior rather than as compilation errors. And the sad fact is that you will probably find them. Again, you can minimize bugs by careful design and careful coding, but no one has found a way to avoid them altogether. Once you've detected a bug, it's time for ***debugging***. You have to track down the cause of the bug in the program's source code and eliminate it. Debugging is a skill that, like other aspects of programming, requires practice to master. So don't be afraid of bugs. Learn from them. One essential debugging skill is the ability to read source code—the ability to put aside preconceptions about what you *think* it does and to follow it the way the computer does—mechanically, step-by-step—to see what it really does. This is hard. I can still remember the time I spent hours looking for a bug only to find that a line of code that I had looked at ten times had a "1" where it should have had an "i", or the time when I wrote a subroutine named `WindowClosing` which would have done exactly what I wanted except that the computer was looking for `windowClosing` (with a lower case "w"). Sometimes it can help to have someone who doesn't share your preconceptions look at your code.

Often, it's a problem just to find the part of the program that contains the error. Most programming environments come with a ***debugger***, which is a program that can help you find bugs. Typically, your program can be run under the control of the debugger. The debugger allows you to set "breakpoints" in your program. A breakpoint is a point in the program where the debugger will pause the program so you can look at the values of the program's variables. The idea is to track down exactly when things start to go wrong during the program's execution. The debugger will also let you execute your program one line at a time, so that you can watch what happens in detail once you know the general area in the program where the bug is lurking.

I will confess that I only occasionally use debuggers myself. A more traditional approach to debugging is to insert ***debugging statements*** into your program. These are output statements that print out information about the state of the program. Typically, a debugging statement would say something like

```
System.out.println("At start of while loop, N = " + N);
```

You need to be able to tell from the output where in your program the output is coming from, and you want to know the value of important variables. Sometimes, you will find that the computer isn't even getting to a part of the program that you think it should be executing. Remember that the goal is to find the first point in the program where the state is not what you expect it to be. That's where the bug is.

And finally, remember the golden rule of debugging: If you are absolutely sure that everything in your program is right, and if it still doesn't work, then one of the things that you are absolutely sure of is wrong.

3.3 The while and do..while Statements

STATEMENTS IN JAVA CAN be either simple statements or compound statements. Simple statements, such as assignment statements and subroutine call statements, are the basic building blocks of a program. Compound statements, such as `while` loops and `if` statements, are used to organize simple statements into complex structures, which are called control structures because they control the order in which the statements are executed. The next five sections explore the details of control structures that are available in Java, starting with the `while` statement and the `do..while` statement in this section. At the same time, we'll look at examples of programming with each control structure and apply the techniques for designing algorithms that were introduced in the previous section.

3.3.1 The while Statement

The `while` statement was already introduced in Section 3.1. A `while` loop has the form

```
while ( ⟨boolean-expression⟩ )
   ⟨statement⟩
```

The ⟨*statement*⟩ can, of course, be a block statement consisting of several statements grouped together between a pair of braces. This statement is called the ***body of the loop***. The body of the loop is repeated as long as the ⟨*boolean-expression*⟩ is true. This boolean expression is called the ***continuation condition***, or more simply the ***test***, of the loop. There are a few points that might need some clarification. What happens if the condition is false in the first place, before the body of the loop is executed even once? In that case, the body of the loop is never executed at all. The body of a while loop can be executed any number of times, including zero. What happens if the condition is true, but it becomes false somewhere in the **middle** of the loop body? Does the loop end as soon as this happens? It doesn't, because the computer continues executing the body of the loop until it gets to the end. Only then does it jump back to the beginning of the loop and test the condition, and only then can the loop end.

Let's look at a typical problem that can be solved using a `while` loop: finding the average of a set of positive integers entered by the user. The average is the sum of the integers, divided by the number of integers. The program will ask the user to enter one integer at a time. It will keep count of the number of integers entered, and it will keep a running total of all the numbers it has read so far. Here is a pseudocode algorithm for the program:

```
Let sum = 0     // The sum of the integers entered by the user.
Let count = 0   // The number of integers entered by the user.
while there are more integers to process:
    Read an integer
    Add it to the sum
    Count it
Divide sum by count to get the average
Print out the average
```

But how can we test whether there are more integers to process? A typical solution is to tell the user to type in zero after all the data have been entered. This will work because we are assuming that all the data are positive numbers, so zero is not a legal data value. The zero is not itself part of the data to be averaged. It's just there to mark the end of the real data. A data value used in this way is sometimes called a ***sentinel value***. So now the test in the while loop becomes "while the input integer is not zero". But there is another problem! The

first time the test is evaluated, before the body of the loop has ever been executed, no integer has yet been read. There is no "input integer" yet, so testing whether the input integer is zero doesn't make sense. So, we have to do something **before** the while loop to make sure that the test makes sense. Setting things up so that the test in a `while` loop makes sense the first time it is executed is called ***priming the loop***. In this case, we can simply read the first integer before the beginning of the loop. Here is a revised algorithm:

```
Let sum = 0
Let count = 0
Read an integer
while the integer is not zero:
    Add the integer to the sum
    Count it
    Read an integer
Divide sum by count to get the average
Print out the average
```

Notice that I've rearranged the body of the loop. Since an integer is read before the loop, the loop has to begin by processing that integer. At the end of the loop, the computer reads a new integer. The computer then jumps back to the beginning of the loop and tests the integer that it has just read. Note that when the computer finally reads the sentinel value, the loop ends before the sentinel value is processed. It is not added to the sum, and it is not counted. This is the way it's supposed to work. The sentinel is not part of the data. The original algorithm, even if it could have been made to work without priming, was incorrect since it would have summed and counted all the integers, including the sentinel. (Since the sentinel is zero, the sum would still be correct, but the count would be off by one. Such so-called ***off-by-one errors*** are very common. Counting turns out to be harder than it looks!)

We can easily turn the algorithm into a complete program. Note that the program cannot use the statement "`average = sum/count;`" to compute the average. Since `sum` and `count` are both variables of type **int**, the value of `sum/count` is an integer. The average should be a real number. We've seen this problem before: we have to convert one of the **int** values to a **double** to force the computer to compute the quotient as a real number. This can be done by type-casting one of the variables to type **double**. The type cast "(double)sum" converts the value of `sum` to a real number, so in the program the average is computed as "`average = ((double)sum) / count;`". Another solution in this case would have been to declare `sum` to be a variable of type **double** in the first place.

One other issue is addressed by the program: If the user enters zero as the first input value, there are no data to process. We can test for this case by checking whether `count` is still equal to zero after the `while` loop. This might seem like a minor point, but a careful programmer should cover all the bases.

Here is the program:

```
/**
 * This program reads a sequence of positive integers input
 * by the user, and it will print out the average of those
 * integers.  The user is prompted to enter one integer at a
 * time.  The user must enter a 0 to mark the end of the
 * data.  (The zero is not counted as part of the data to
 * be averaged.)  The program does not check whether the
 * user's input is positive, so it will actually add up
 * both positive and negative input values.
```

```
 */
public class ComputeAverage {

   public static void main(String[] args) {

      int inputNumber;   // One of the integers input by the user.
      int sum;           // The sum of the positive integers.
      int count;         // The number of positive integers.
      double average;    // The average of the positive integers.

      /* Initialize the summation and counting variables. */

      sum = 0;
      count = 0;

      /* Read and process the user's input. */

      TextIO.put("Enter your first positive integer: ");
      inputNumber = TextIO.getlnInt();

      while (inputNumber != 0) {
         sum += inputNumber;   // Add inputNumber to running sum.
         count++;              // Count the input by adding 1 to count.
         TextIO.put("Enter your next positive integer, or 0 to end: ");
         inputNumber = TextIO.getlnInt();
      }

      /* Display the result. */

      if (count == 0) {
         TextIO.putln("You didn't enter any data!");
      }
      else {
         average = ((double)sum) / count;
         TextIO.putln();
         TextIO.putln("You entered " + count + " positive integers.");
         TextIO.putf("Their average is %1.3f.\n", average);
      }

   } // end main()

} // end class ComputeAverage
```

3.3.2 The do..while Statement

Sometimes it is more convenient to test the continuation condition at the end of a loop, instead of at the beginning, as is done in the `while` loop. The `do..while` statement is very similar to the `while` statement, except that the word "while," along with the condition that it tests, has been moved to the end. The word "do" is added to mark the beginning of the loop. A `do..while` statement has the form

```
do
    ⟨statement⟩
while ( ⟨boolean-expression⟩ );
```

or, since, as usual, the ⟨*statement*⟩ can be a block,

```
do {
    ⟨statements⟩
} while ( ⟨boolean-expression⟩ );
```

Note the semicolon, ';', at the very end. This semicolon is part of the statement, just as the semicolon at the end of an assignment statement or declaration is part of the statement. Omitting it is a syntax error. (More generally, **every** statement in Java ends either with a semicolon or a right brace, '}'.)

To execute a `do` loop, the computer first executes the body of the loop—that is, the statement or statements inside the loop—and then it evaluates the boolean expression. If the value of the expression is `true`, the computer returns to the beginning of the `do` loop and repeats the process; if the value is `false`, it ends the loop and continues with the next part of the program. Since the condition is not tested until the end of the loop, the body of a `do` loop is always executed at least once.

For example, consider the following pseudocode for a game-playing program. The `do` loop makes sense here instead of a `while` loop because with the `do` loop, you know there will be at least one game. Also, the test that is used at the end of the loop wouldn't even make sense at the beginning:

```
do {
   Play a Game
   Ask user if he wants to play another game
   Read the user's response
} while ( the user's response is yes );
```

Let's convert this into proper Java code. Since I don't want to talk about game playing at the moment, let's say that we have a class named `Checkers`, and that the `Checkers` class contains a static member subroutine named `playGame()` that plays one game of checkers against the user. Then, the pseudocode "Play a game" can be expressed as the subroutine call statement "`Checkers.playGame();`". We need a variable to store the user's response. The *TextIO* class makes it convenient to use a **boolean** variable to store the answer to a yes/no question. The input function `TextIO.getlnBoolean()` allows the user to enter the value as "yes" or "no". "Yes" is considered to be `true`, and "no" is considered to be `false`. So, the algorithm can be coded as

```
boolean wantsToContinue;  // True if user wants to play again.
do {
   Checkers.playGame();
   TextIO.put("Do you want to play again? ");
   wantsToContinue = TextIO.getlnBoolean();
} while (wantsToContinue == true);
```

When the value of the **boolean** variable is set to `false`, it is a signal that the loop should end. When a **boolean** variable is used in this way—as a signal that is set in one part of the program and tested in another part—it is sometimes called a ***flag*** or ***flag variable*** (in the sense of a signal flag).

By the way, a more-than-usually-pedantic programmer would sneer at the test "`while (wantsToContinue == true)`". This test is exactly equivalent to "`while (wantsToContinue)`". Testing whether "`wantsToContinue == true`" is true amounts to the same thing as testing whether "`wantsToContinue`" is true. A little less offensive is an expression of the form "`flag == false`", where `flag` is a boolean variable. The value of "`flag == false`" is exactly the same as the value of "`!flag`", where `!` is the boolean negation operator. So

you can write "`while (!flag)`" instead of "`while (flag == false)`", and you can write "`if (!flag)`" instead of "`if (flag == false)`".

Although a `do..while` statement is sometimes more convenient than a `while` statement, having two kinds of loops does not make the language more powerful. Any problem that can be solved using `do..while` loops can also be solved using only `while` statements, and vice versa. In fact, if ⟨*doSomething*⟩ represents any block of program code, then

```
do {
    ⟨doSomething⟩
} while ( ⟨boolean-expression⟩ );
```

has exactly the same effect as

```
⟨doSomething⟩
while ( ⟨boolean-expression⟩ ) {
    ⟨doSomething⟩
}
```

Similarly,

```
while ( ⟨boolean-expression⟩ ) {
    ⟨doSomething⟩
}
```

can be replaced by

```
if ( ⟨boolean-expression⟩ ) {
   do {
       ⟨doSomething⟩
   } while ( ⟨boolean-expression⟩ );
}
```

without changing the meaning of the program in any way.

3.3.3 break and continue

The syntax of the `while` and `do..while` loops allows you to test the continuation condition at either the beginning of a loop or at the end. Sometimes, it is more natural to have the test in the middle of the loop, or to have several tests at different places in the same loop. Java provides a general method for breaking out of the middle of any loop. It's called the `break` statement, which takes the form

```
break;
```

When the computer executes a `break` statement in a loop, it will immediately jump out of the loop. It then continues on to whatever follows the loop in the program. Consider for example:

```
while (true) {  // looks like it will run forever!
   TextIO.put("Enter a positive number: ");
   N = TextIO.getlnInt();
   if (N > 0)   // input is OK; jump out of loop
      break;
   TextIO.putln("Your answer must be > 0.");
}
// continue here after break
```

If the number entered by the user is greater than zero, the `break` statement will be executed and the computer will jump out of the loop. Otherwise, the computer will print out "Your answer must be > 0." and will jump back to the start of the loop to read another input value.

The first line of this loop, "`while (true)`" might look a bit strange, but it's perfectly legitimate. The condition in a `while` loop can be any boolean-valued expression. The computer evaluates this expression and checks whether the value is `true` or `false`. The boolean literal "`true`" is just a boolean expression that always evaluates to true. So "`while (true)`" can be used to write an infinite loop, or one that will be terminated by a `break` statement.

A `break` statement terminates the loop that immediately encloses the `break` statement. It is possible to have ***nested*** loops, where one loop statement is contained inside another. If you use a `break` statement inside a nested loop, it will only break out of that loop, not out of the loop that contains the nested loop. There is something called a ***labeled break*** statement that allows you to specify which loop you want to break. This is not very common, so I will go over it quickly. Labels work like this: You can put a ***label*** in front of any loop. A label consists of a simple identifier followed by a colon. For example, a `while` with a label might look like "`mainloop: while...`". Inside this loop you can use the labeled break statement "`break mainloop;`" to break out of the labeled loop. For example, here is a code segment that checks whether two strings, `s1` and `s2`, have a character in common. If a common character is found, the value of the flag variable `nothingInCommon` is set to `false`, and a labeled break is used to end the processing at that point:

```
boolean nothingInCommon;
nothingInCommon = true;  // Assume s1 and s2 have no chars in common.
int i,j;  // Variables for iterating through the chars in s1 and s2.

i = 0;
bigloop: while (i < s1.length()) {
   j = 0;
   while (j < s2.length()) {
      if (s1.charAt(i) == s2.charAt(j)) { // s1 and s2 have a common char.
          nothingInCommon = false;
          break bigloop;  // break out of BOTH loops
      }
      j++;  // Go on to the next char in s2.
   }
   i++;  //Go on to the next char in s1.
}
```

The `continue` statement is related to `break`, but less commonly used. A `continue` statement tells the computer to skip the rest of the current iteration of the loop. However, instead of jumping out of the loop altogether, it jumps back to the beginning of the loop and continues with the next iteration (including evaluating the loop's continuation condition to see whether any further iterations are required). As with `break`, when a `continue` is in a nested loop, it will continue the loop that directly contains it; a "labeled continue" can be used to continue the containing loop instead.

`break` and `continue` can be used in `while` loops and `do..while` loops. They can also be used in `for` loops, which are covered in the next section. In Section 3.6, we'll see that `break` can also be used to break out of a `switch` statement. A `break` can occur inside an `if` statement, but in that case, it does **not** mean to break out of the `if`. Instead, it breaks out of the loop or `switch` statement that contains the `if` statement. If the `if` statement is not contained inside a

loop or `switch`, then the `if` statement cannot legally contain a `break`. A similar consideration applies to `continue` statements inside `ifs`.

3.4 The for Statement

We turn in this section to another type of loop, the `for` statement. Any `for` loop is equivalent to some `while` loop, so the language doesn't get any additional power by having `for` statements. But for a certain type of problem, a `for` loop can be easier to construct and easier to read than the corresponding `while` loop. It's quite possible that in real programs, `for` loops actually outnumber `while` loops.

3.4.1 For Loops

The `for` statement makes a common type of while loop easier to write. Many while loops have the general form:

```
⟨initialization⟩
while ( ⟨continuation-condition⟩ ) {
    ⟨statements⟩
    ⟨update⟩
}
```

For example, consider this example, copied from an example in Section 3.2:

```
years = 0;  // initialize the variable years
while ( years < 5 ) {   // condition for continuing loop

    interest = principal * rate;    //
    principal += interest;          // do three statements
    System.out.println(principal);  //

    years++;   // update the value of the variable, years
}
```

This loop can be written as the following equivalent `for` statement:

```
for ( years = 0;  years < 5;  years++ ) {
   interest = principal * rate;
   principal += interest;
   System.out.println(principal);
}
```

The initialization, continuation condition, and updating have all been combined in the first line of the `for` loop. This keeps everything involved in the "control" of the loop in one place, which helps make the loop easier to read and understand. The `for` loop is executed in exactly the same way as the original code: The initialization part is executed once, before the loop begins. The continuation condition is executed before each execution of the loop, and the loop ends when this condition is `false`. The update part is executed at the end of each execution of the loop, just before jumping back to check the condition.

The formal syntax of the `for` statement is as follows:

```
for ( ⟨initialization⟩; ⟨continuation-condition⟩; ⟨update⟩ )
     ⟨statement⟩
```

or, using a block statement:

```
for ( ⟨initialization⟩; ⟨continuation-condition⟩; ⟨update⟩ ) {
     ⟨statements⟩
}
```

The ⟨*continuation-condition*⟩ must be a boolean-valued expression. The ⟨*initialization*⟩ is usually a declaration or an assignment statement, but it can be any expression that would be allowed as a statement in a program. The ⟨*update*⟩ can be any expression statement, but is usually an increment, a decrement, or an assignment statement. Any of the three can be empty. If the continuation condition is empty, it is treated as if it were "`true`," so the loop will be repeated forever or until it ends for some other reason, such as a `break` statement. (Some people like to begin an infinite loop with "`for (;;)`" instead of "`while (true)`".)

Usually, the initialization part of a `for` statement assigns a value to some variable, and the update changes the value of that variable with an assignment statement or with an increment or decrement operation. The value of the variable is tested in the continuation condition, and the loop ends when this condition evaluates to `false`. A variable used in this way is called a ***loop control variable***. In the `for` statement given above, the loop control variable is `years`.

Certainly, the most common type of `for` loop is the ***counting loop***, where a loop control variable takes on all integer values between some minimum and some maximum value. A counting loop has the form

```
for ( ⟨variable⟩ = ⟨min⟩;  ⟨variable⟩ <= ⟨max⟩; ⟨variable⟩++ ) {
     ⟨statements⟩
}
```

where ⟨*min*⟩ and ⟨*max*⟩ are integer-valued expressions (usually constants). The ⟨*variable*⟩ takes on the values ⟨*min*⟩, ⟨*min*⟩+1, ⟨*min*⟩+2, ..., ⟨*max*⟩. The value of the loop control variable is often used in the body of the loop. The `for` loop at the beginning of this section is a counting loop in which the loop control variable, `years`, takes on the values 1, 2, 3, 4, 5. Here is an even simpler example, in which the numbers 1, 2, ..., 10 are displayed on standard output:

```
for ( N = 1 ;  N <= 10 ;  N++ )
   System.out.println( N );
```

For various reasons, Java programmers like to start counting at 0 instead of 1, and they tend to use a "`<`" in the condition, rather than a "`<=`". The following variation of the above loop prints out the ten numbers 0, 1, 2, ..., 9:

```
for ( N = 0 ;  N < 10 ;  N++ )
   System.out.println( N );
```

Using `<` instead of `<=` in the test, or vice versa, is a common source of off-by-one errors in programs. You should always stop and think, Do I want the final value to be processed or not?

It's easy to count down from 10 to 1 instead of counting up. Just start with 10, decrement the loop control variable instead of incrementing it, and continue as long as the variable is greater than or equal to one.

```
for ( N = 10 ;  N >= 1 ;  N-- )
   System.out.println( N );
```

Now, in fact, the official syntax of a `for` statement actually allows both the initialization part and the update part to consist of several expressions, separated by commas. So we can even count up from 1 to 10 and count down from 10 to 1 at the same time!

```
for ( i=1, j=10;  i <= 10;  i++, j-- ) {
   System.out.printf("%5d", i); // Output i in a 5-character wide column.
   System.out.printf("%5d", j); // Output j in a 5-character column
   System.out.println();       //     and end the line.
}
```

As a final introductory example, let's say that we want to use a `for` loop that prints out just the even numbers between 2 and 20, that is: 2, 4, 6, 8, 10, 12, 14, 16, 18, 20. There are several ways to do this. Just to show how even a very simple problem can be solved in many ways, here are four different solutions (three of which would get full credit):

```
(1)   // There are 10 numbers to print.
      // Use a for loop to count 1, 2,
      // ..., 10.  The numbers we want
      // to print are 2*1, 2*2, ... 2*10.

      for (N = 1; N <= 10; N++) {
         System.out.println( 2*N );
      }


(2)   // Use a for loop that counts
      // 2, 4, ..., 20 directly by
      // adding 2 to N each time through
      // the loop.

      for (N = 2; N <= 20; N = N + 2) {
         System.out.println( N );
      }


(3)   // Count off all the numbers
      // 2, 3, 4, ..., 19, 20, but
      // only print out the numbers
      // that are even.

      for (N = 2; N <= 20; N++) {
         if ( N % 2 == 0 ) // is N even?
            System.out.println( N );
      }


(4)   // Irritate the professor with
      // a solution that follows the
      // letter of this silly assignment
      // while making fun of it.

      for (N = 1; N <= 1; N++) {
         System.out.println("2 4 6 8 10 12 14 16 18 20");
      }
```

Perhaps it is worth stressing one more time that a `for` statement, like any statement, never occurs on its own in a real program. A statement must be inside the `main` routine of a program or inside some other subroutine. And that subroutine must be defined inside a class. I should also remind you that every variable must be declared before it can be used, and that includes the loop control variable in a `for` statement. In all the examples that you have seen so far in this section, the loop control variables should be declared to be of type **int**. It is not required

that a loop control variable be an integer. Here, for example, is a `for` loop in which the variable, `ch`, is of type **char**, using the fact that the `++` operator can be applied to characters as well as to numbers:

```
// Print out the alphabet on one line of output.
char ch;  // The loop control variable;
          //       one of the letters to be printed.
for ( ch = 'A';  ch <= 'Z';  ch++ )
    System.out.print(ch);
System.out.println();
```

3.4.2 Example: Counting Divisors

Let's look at a less trivial problem that can be solved with a `for` loop. If `N` and `D` are positive integers, we say that `D` is a ***divisor*** of `N` if the remainder when `D` is divided into `N` is zero. (Equivalently, we could say that `N` is an even multiple of `D`.) In terms of Java programming, `D` is a divisor of `N` if `N % D` is zero.

Let's write a program that inputs a positive integer, `N`, from the user and computes how many different divisors `N` has. The numbers that could possibly be divisors of `N` are 1, 2, . . . , `N`. To compute the number of divisors of `N`, we can just test each possible divisor of `N` and count the ones that actually do divide `N` evenly. In pseudocode, the algorithm takes the form

```
Get a positive integer, N, from the user
Let divisorCount = 0
for each number, testDivisor, in the range from 1 to N:
    if testDivisor is a divisor of N:
        Count it by adding 1 to divisorCount
Output the count
```

This algorithm displays a common programming pattern that is used when some, but not all, of a sequence of items are to be processed. The general pattern is

```
for each item in the sequence:
   if the item passes the test:
       process it
```

The `for` loop in our divisor-counting algorithm can be translated into Java code as

```
for (testDivisor = 1; testDivisor <= N; testDivisor++) {
   if ( N % testDivisor == 0 )
      divisorCount++;
}
```

On a modern computer, this loop can be executed very quickly. It is not impossible to run it even for the largest legal **int** value, 2147483647. (If you wanted to run it for even larger values, you could use variables of type **long** rather than **int**.) However, it does take a significant amount of time for very large numbers. So when I implemented this algorithm, I decided to output a dot every time the computer has tested one million possible divisors. In the improved version of the program, there are two types of counting going on. We have to count the number of divisors and we also have to count the number of possible divisors that have been tested. So the program needs two counters. When the second counter reaches 1000000, the program outputs a '.' and resets the counter to zero so that we can start counting the next group of one million. Reverting to pseudocode, the algorithm now looks like

```
Get a positive integer, N, from the user
Let divisorCount = 0  // Number of divisors found.
Let numberTested = 0  // Number of possible divisors tested
                      //       since the last period was output.
for each number, testDivisor, in the range from 1 to N:
    if testDivisor is a divisor of N:
        Count it by adding 1 to divisorCount
    Add 1 to numberTested
    if numberTested is 1000000:
        print out a '.'
        Reset numberTested to 0
Output the count
```

Finally, we can translate the algorithm into a complete Java program:

```
/**
 * This program reads a positive integer from the user.
 * It counts how many divisors that number has, and
 * then it prints the result.
 */

public class CountDivisors {

   public static void main(String[] args) {

      int N;  // A positive integer entered by the user.
              // Divisors of this number will be counted.

      int testDivisor;  // A number between 1 and N that is a
                        // possible divisor of N.

      int divisorCount;  // Number of divisors of N that have been found.

      int numberTested;  // Used to count how many possible divisors
                         // of N have been tested.  When the number
                         // reaches 1000000, a period is output and
                         // the value of numberTested is reset to zero.

      /* Get a positive integer from the user. */

      while (true) {
         System.out.print("Enter a positive integer: ");
         N = TextIO.getlnInt();
         if (N > 0)
            break;
         System.out.println("That number is not positive.  Please try again.");
      }

      /* Count the divisors, printing a "." after every 1000000 tests. */

      divisorCount = 0;
      numberTested = 0;

      for (testDivisor = 1; testDivisor <= N; testDivisor++) {
         if ( N % testDivisor == 0 )
            divisorCount++;
         numberTested++;
         if (numberTested == 1000000) {
            System.out.print('.');
```

```
            numberTested = 0;
         }
      }

      /* Display the result. */

      System.out.println();
      System.out.println("The number of divisors of " + N
                         + " is " + divisorCount);

   } // end main()

} // end class CountDivisors
```

3.4.3 Nested for Loops

Control structures in Java are statements that contain statements. In particular, control structures can contain control structures. You've already seen several examples of `if` statements inside loops, and one example of a `while` loop inside another `while`, but any combination of one control structure inside another is possible. We say that one structure is ***nested*** inside another. You can even have multiple levels of nesting, such as a `while` loop inside an `if` statement inside another `while` loop. The syntax of Java does not set a limit on the number of levels of nesting. As a practical matter, though, it's difficult to understand a program that has more than a few levels of nesting.

Nested `for` loops arise naturally in many algorithms, and it is important to understand how they work. Let's look at a couple of examples. First, consider the problem of printing out a multiplication table like this one:

```
 1   2   3   4   5   6   7   8   9  10  11  12
 2   4   6   8  10  12  14  16  18  20  22  24
 3   6   9  12  15  18  21  24  27  30  33  36
 4   8  12  16  20  24  28  32  36  40  44  48
 5  10  15  20  25  30  35  40  45  50  55  60
 6  12  18  24  30  36  42  48  54  60  66  72
 7  14  21  28  35  42  49  56  63  70  77  84
 8  16  24  32  40  48  56  64  72  80  88  96
 9  18  27  36  45  54  63  72  81  90  99 108
10  20  30  40  50  60  70  80  90 100 110 120
11  22  33  44  55  66  77  88  99 110 121 132
12  24  36  48  60  72  84  96 108 120 132 144
```

The data in the table are arranged into 12 rows and 12 columns. The process of printing them out can be expressed in a pseudocode algorithm as

```
for each rowNumber = 1, 2, 3, ..., 12:
   Print the first twelve multiples of rowNumber on one line
   Output a carriage return
```

The first step in the `for` loop can itself be expressed as a `for` loop. We can expand "Print the first twelve multiples of `rowNumber` on one line" as:

```
for N = 1, 2, 3, ..., 12:
   Print N * rowNumber
```

so a refined algorithm for printing the table has one `for` loop nested inside another:

```
for each rowNumber = 1, 2, 3, ..., 12:
   for N = 1, 2, 3, ..., 12:
      Print N * rowNumber
   Output a carriage return
```

We want to print the output in neat columns, with each output number taking up four spaces. This can be done using formatted output with format specifier `%4d`. Assuming that `rowNumber` and `N` have been declared to be variables of type **int**, the algorithm can be expressed in Java as

```
for ( rowNumber = 1;  rowNumber <= 12;  rowNumber++ ) {
   for ( N = 1;  N <= 12;  N++ ) {
               // print in 4-character columns
      System.out.printf( "%4d", N * rowNumber );  // No carriage return !
   }
   System.out.println();  // Add a carriage return at end of the line.
}
```

This section has been weighed down with lots of examples of numerical processing. For our next example, let's do some text processing. Consider the problem of finding which of the 26 letters of the alphabet occur in a given string. For example, the letters that occur in "Hello World" are D, E, H, L, O, R, and W. More specifically, we will write a program that will list all the letters contained in a string and will also count the number of different letters. The string will be input by the user. Let's start with a pseudocode algorithm for the program.

```
Ask the user to input a string
Read the response into a variable, str
Let count = 0  (for counting the number of different letters)
for each letter of the alphabet:
   if the letter occurs in str:
      Print the letter
      Add 1 to count
Output the count
```

Since we want to process the entire line of text that is entered by the user, we'll use `TextIO.getln()` to read it. The line of the algorithm that reads "for each letter of the alphabet" can be expressed as "`for (letter='A'; letter<='Z'; letter++)`". But the body of this `for` loop needs more thought. How do we check whether the given letter, `letter`, occurs in `str`? One idea is to look at each character in the string in turn, and check whether that character is equal to `letter`. We can get the `i`-th character of `str` with the function call `str.charAt(i)`, where `i` ranges from 0 to `str.length() - 1`.

One more difficulty: A letter such as 'A' can occur in `str` in either upper or lower case, 'A' or 'a'. We have to check for both of these. But we can avoid this difficulty by converting `str` to upper case before processing it. Then, we only have to check for the upper case letter. We can now flesh out the algorithm fully:

```
Ask the user to input a string
Read the response into a variable, str
Convert str to upper case
Let count = 0
for letter = 'A', 'B', ..., 'Z':
    for i = 0, 1, ..., str.length()-1:
        if letter == str.charAt(i):
            Print letter
            Add 1 to count
```

```
            break  // jump out of the loop, to avoid counting letter twice
Output the count
```

Note the use of `break` in the nested `for` loop. It is required to avoid printing or counting a given letter more than once (in the case where it occurs more than once in the string). The `break` statement breaks out of the inner `for` loop, but not the outer `for` loop. Upon executing the `break`, the computer continues the outer loop with the next value of `letter`. You should try to figure out exactly what `count` would be at the end of this program, if the `break` statement were omitted.

Here is the complete program:

```
/**
 * This program reads a line of text entered by the user.
 * It prints a list of the letters that occur in the text,
 * and it reports how many different letters were found.
 */

public class ListLetters {

   public static void main(String[] args) {

      String str;  // Line of text entered by the user.
      int count;   // Number of different letters found in str.
      char letter; // A letter of the alphabet.

      TextIO.putln("Please type in a line of text.");
      str = TextIO.getln();

      str = str.toUpperCase();

      count = 0;
      TextIO.putln("Your input contains the following letters:");
      TextIO.putln();
      TextIO.put("   ");
      for ( letter = 'A'; letter <= 'Z'; letter++ ) {
          int i;  // Position of a character in str.
          for ( i = 0; i < str.length(); i++ ) {
              if ( letter == str.charAt(i) ) {
                  TextIO.put(letter);
                  TextIO.put(' ');
                  count++;
                  break;
              }
          }
      }

      TextIO.putln();
      TextIO.putln();
      TextIO.putln("There were " + count + " different letters.");

   } // end main()

} // end class ListLetters
```

In fact, there is actually an easier way to determine whether a given letter occurs in a string, `str`. The built-in function `str.indexOf(letter)` will return `-1` if `letter` does **not** occur in the string. It returns a number greater than or equal to zero if it does occur. So, we could check whether `letter` occurs in `str` simply by checking "`if (str.indexOf(letter) >= 0)`". If we used this technique in the above program, we wouldn't need a nested `for` loop. This gives you a preview of how subroutines can be used to deal with complexity.

3.4.4 Enums and for-each Loops

Java 5.0 introduced a new "enhanced" form of the `for` loop that is designed to be convenient for processing data structures. A ***data structure*** is a collection of data items, considered as a unit. For example, a ***list*** is a data structure that consists simply of a sequence of items. The enhanced `for` loop makes it easy to apply the same processing to every element of a list or other data structure. Data structures are a major topic in computer science, but we won't encounter them in any serious way until Chapter 7. However, one of the applications of the enhanced `for` loop is to `enum` types, and so we consider it briefly here. (Enums were introduced in Subsection 2.3.3.)

The enhanced for loop can be used to perform the same processing on each of the enum constants that are the possible values of an enumerated type. The syntax for doing this is:

```
for ( ⟨enum-type-name⟩ ⟨variable-name⟩ : ⟨enum-type-name⟩.values() )
   ⟨statement⟩
```

or

```
for ( ⟨enum-type-name⟩ ⟨variable-name⟩ : ⟨enum-type-name⟩.values() ) {
   ⟨statements⟩
}
```

If *MyEnum* is the name of any enumerated type, then `MyEnum.values()` is a function call that returns a list containing all of the values of the enum. (`values()` is a static member function in *MyEnum* and of any other `enum`.) For this enumerated type, the `for` loop would have the form:

```
for ( MyEnum ⟨variable-name⟩ : MyEnum.values() )
   ⟨statement⟩
```

The intent of this is to execute the ⟨*statement*⟩ once for each of the possible values of the *MyEnum* type. The ⟨*variable-name*⟩ is the loop control variable. In the ⟨*statement*⟩, it represents the enumerated type value that is currently being processed. This variable should **not** be declared before the `for` loop; it is essentially being declared in the loop itself.

To give a concrete example, suppose that the following enumerated type has been defined to represent the days of the week:

```
enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY }
```

Then we could write:

```
for ( Day d : Day.values() ) {
   System.out.print( d );
   System.out.print(" is day number ");
   System.out.println( d.ordinal() );
}
```

`Day.values()` represents the list containing the seven constants that make up the enumerated type. The first time through this loop, the value of `d` would be the first enumerated type value `Day.MONDAY`, which has ordinal number 0, so the output would be "MONDAY is day number 0". The second time through the loop, the value of `d` would be `Day.TUESDAY`, and so on through `Day.SUNDAY`. The body of the loop is executed once for each item in the list `Day.values()`, with `d` taking on each of those values in turn. The full output from this loop would be:

```
MONDAY is day number 0
TUESDAY is day number 1
WEDNESDAY is day number 2
THURSDAY is day number 3
FRIDAY is day number 4
SATURDAY is day number 5
SUNDAY is day number 6
```

Since the intent of the enhanced for loop is to do something "for each" item in a data structure, it is often called a ***for-each loop***. The syntax for this type of loop is unfortunate. It would be better if it were written something like "`foreach Day d in Day.values()`", which conveys the meaning much better and is similar to the syntax used in other programming languages for similar types of loops. It's helpful to think of the colon (:) in the loop as meaning "in."

3.5 The if Statement

The first of the two branching statements in Java is the `if` statement, which you have already seen in Section 3.1. It takes the form

```
if (⟨boolean-expression⟩)
     ⟨statement-1⟩
else
     ⟨statement-2⟩
```

As usual, the statements inside an `if` statement can be blocks. The `if` statement represents a two-way branch. The `else` part of an `if` statement—consisting of the word "else" and the statement that follows it—can be omitted.

3.5.1 The Dangling else Problem

Now, an `if` statement is, in particular, a statement. This means that either ⟨*statement-1*⟩ or ⟨*statement-2*⟩ in the above `if` statement can itself be an `if` statement. A problem arises, however, if ⟨*statement-1*⟩ is an `if` statement that has no `else` part. This special case is effectively forbidden by the syntax of Java. Suppose, for example, that you type

```
if ( x > 0 )
    if (y > 0)
       System.out.println("First case");
else
    System.out.println("Second case");
```

Now, remember that the way you've indented this doesn't mean anything at all to the computer. You might think that the `else` part is the second half of your "`if (x > 0)`" statement, but the rule that the computer follows attaches the `else` to "`if (y > 0)`", which is closer. That is, the computer reads your statement as if it were formatted:

```
if ( x > 0 )
    if (y > 0)
       System.out.println("First case");
    else
        System.out.println("Second case");
```

You can force the computer to use the other interpretation by enclosing the nested `if` in a block:

```
if ( x > 0 ) {
    if (y > 0)
       System.out.println("First case");
}
else
    System.out.println("Second case");
```

These two `if` statements have different meanings: In the case when `x <= 0`, the first statement doesn't print anything, but the second statement prints "Second case".

3.5.2 The if...else if Construction

Much more interesting than this technicality is the case where ⟨*statement-2*⟩, the `else` part of the `if` statement, is itself an `if` statement. The statement would look like this (perhaps without the final else part):

```
if (⟨boolean-expression-1⟩)
     ⟨statement-1⟩
else
     if (⟨boolean-expression-2⟩)
         ⟨statement-2⟩
     else
         ⟨statement-3⟩
```

However, since the computer doesn't care how a program is laid out on the page, this is almost always written in the format:

```
if (⟨boolean-expression-1⟩)
     ⟨statement-1⟩
else if (⟨boolean-expression-2⟩)
     ⟨statement-2⟩
else
     ⟨statement-3⟩
```

You should think of this as a single statement representing a three-way branch. When the computer executes this, one and only one of the three statements—⟨*statement-1*⟩, ⟨*statement-2*⟩, or ⟨*statement-3*⟩—will be executed. The computer starts by evaluating ⟨*boolean-expression-1*⟩. If it is `true`, the computer executes ⟨*statement-1*⟩ and then jumps all the way to the end of the outer if statement, skipping the other two ⟨*statement*⟩s. If ⟨*boolean-expression-1*⟩ is `false`, the computer skips ⟨*statement-1*⟩ and executes the second, nested if statement. To do this, it tests the value of ⟨*boolean-expression-2*⟩ and uses it to decide between ⟨*statement-2*⟩ and ⟨*statement-3*⟩.

Here is an example that will print out one of three different messages, depending on the value of a variable named `temperature`:

```
if (temperature < 50)
   System.out.println("It's cold.");
else if (temperature < 80)
   System.out.println("It's nice.");
else
   System.out.println("It's hot.");
```

If `temperature` is, say, 42, the first test is `true`. The computer prints out the message "It's cold", and skips the rest—without even evaluating the second condition. For a temperature of 75, the first test is `false`, so the computer goes on to the second test. This test is `true`, so the computer prints "It's nice" and skips the rest. If the temperature is 173, both of the tests evaluate to `false`, so the computer says "It's hot" (unless its circuits have been fried by the heat, that is).

You can go on stringing together "else-if's" to make multi-way branches with any number of cases:

```
if (⟨boolean-expression-1⟩)
     ⟨statement-1⟩
else if (⟨boolean-expression-2⟩)
     ⟨statement-2⟩
else if (⟨boolean-expression-3⟩)
     ⟨statement-3⟩
  .
  . // (more cases)
  .
else if (⟨boolean-expression-N⟩)
     ⟨statement-N⟩
else
     ⟨statement-(N+1)⟩
```

The computer evaluates boolean expressions one after the other until it comes to one that is `true`. It executes the associated statement and skips the rest. If none of the boolean expressions evaluate to `true`, then the statement in the `else` part is executed. This statement is called a multi-way branch because only one of the statements will be executed. The final `else` part can be omitted. In that case, if all the boolean expressions are false, none of the statements are executed. Of course, each of the statements can be a block, consisting of a number of statements enclosed between `{` and `}`. (Admittedly, there is lot of syntax here; as you study and practice, you'll become comfortable with it.)

3.5.3 If Statement Examples

As an example of using `if` statements, lets suppose that `x`, `y`, and `z` are variables of type **int**, and that each variable has already been assigned a value. Consider the problem of printing out the values of the three variables in increasing order. For examples, if the values are 42, 17, and 20, then the output should be in the order 17, 20, 42.

One way to approach this is to ask, where does `x` belong in the list? It comes first if it's less than both `y` and `z`. It comes last if it's greater than both `y` and `z`. Otherwise, it comes in the middle. We can express this with a 3-way `if` statement, but we still have to worry about the order in which `y` and `z` should be printed. In pseudocode,

```
if (x < y && x < z) {
    output x, followed by y and z in their correct order
```

```
}
else if (x > y && x > z) {
    output y and z in their correct order, followed by x
}
else {
    output x in between y and z in their correct order
}
```

Determining the relative order of `y` and `z` requires another `if` statement, so this becomes

```
if (x < y && x < z) {        // x comes first
    if (y < z)
       System.out.println( x + " " + y + " " + z );
    else
       System.out.println( x + " " + z + " " + y );
}
else if (x > y && x > z) {   // x comes last
    if (y < z)
       System.out.println( y + " " + z + " " + x );
    else
       System.out.println( z + " " + y + " " + x );
}
else {                       // x in the middle
    if (y < z)
       System.out.println( y + " " + x + " " + z);
    else
       System.out.println( z + " " + x + " " + y);
}
```

You might check that this code will work correctly even if some of the values are the same. If the values of two variables are the same, it doesn't matter which order you print them in.

Note, by the way, that even though you can say in English "if x is less than y and z," you can't say in Java "`if (x < y && z)`". The `&&` operator can only be used between boolean values, so you have to make separate tests, `x<y` and `x<z`, and then combine the two tests with `&&`.

There is an alternative approach to this problem that begins by asking, "which order should `x` and `y` be printed in?" Once that's known, you only have to decide where to stick in `z`. This line of thought leads to different Java code:

```
if ( x < y ) {  // x comes before y
   if ( z < x )   // z comes first
      System.out.println( z + " " + x + " " + y);
   else if ( z > y )   // z comes last
      System.out.println( x + " " + y + " " + z);
   else   // z is in the middle
      System.out.println( x + " " + z + " " + y);
}
else {          // y comes before x
   if ( z < y )   // z comes first
      System.out.println( z + " " + y + " " + x);
   else if ( z > x )  // z comes last
      System.out.println( y + " " + x + " " + z);
   else  // z is in the middle
      System.out.println( y + " " + z + " " + x);
}
```

Once again, we see how the same problem can be solved in many different ways. The two approaches to this problem have not exhausted all the possibilities. For example, you might start by testing whether `x` is greater than `y`. If so, you could swap their values. Once you've done that, you know that `x` should be printed before `y`.

* * *

Finally, let's write a complete program that uses an `if` statement in an interesting way. I want a program that will convert measurements of length from one unit of measurement to another, such as miles to yards or inches to feet. So far, the problem is extremely underspecified. Let's say that the program will only deal with measurements in inches, feet, yards, and miles. It would be easy to extend it later to deal with other units. The user will type in a measurement in one of these units, such as "17 feet" or "2.73 miles". The output will show the length in terms of **each** of the four units of measure. (This is easier than asking the user which units to use in the output.) An outline of the process is

```
Read the user's input measurement and units of measure
Express the measurement in inches, feet, yards, and miles
Display the four results
```

The program can read both parts of the user's input from the same line by using `TextIO.getDouble()` to read the numerical measurement and `TextIO.getlnWord()` to read the unit of measure. The conversion into different units of measure can be simplified by first converting the user's input into inches. From there, the number of inches can easily be converted into feet, yards, and miles. Before converting into inches, we have to test the input to determine which unit of measure the user has specified:

```
Let measurement = TextIO.getDouble()
Let units = TextIO.getlnWord()
if the units are inches
   Let inches = measurement
else if the units are feet
   Let inches = measurement * 12         // 12 inches per foot
else if the units are yards
   Let inches = measurement * 36         // 36 inches per yard
else if the units are miles
   Let inches = measurement * 12 * 5280  // 5280 feet per mile
else
   The units are illegal!
   Print an error message and stop processing
Let feet = inches / 12.0
Let yards = inches / 36.0
Let miles = inches / (12.0 * 5280.0)
Display the results
```

Since `units` is a *String*, we can use `units.equals("inches")` to check whether the specified unit of measure is "inches". However, it would be nice to allow the units to be specified as "inch" or abbreviated to "in". To allow these three possibilities, we can check `if (units.equals("inches") || units.equals("inch") || units.equals("in"))`. It would also be nice to allow upper case letters, as in "Inches" or "IN". We can do this by converting `units` to lower case before testing it or by substituting the function `units.equalsIgnoreCase` for `units.equals`.

In my final program, I decided to make things more interesting by allowing the user to repeat the process of entering a measurement and seeing the results of the conversion for each

measurement. The program will end only when the user inputs 0. To do this, I just have to wrap the above algorithm inside a `while` loop, and make sure that the loop ends when the user inputs a 0. Here's the complete program:

```
/**
 * This program will convert measurements expressed in inches,
 * feet, yards, or miles into each of the possible units of
 * measure.  The measurement is input by the user, followed by
 * the unit of measure.  For example:  "17 feet", "1 inch", or
 * "2.73 mi".  Abbreviations in, ft, yd, and mi are accepted.
 * The program will continue to read and convert measurements
 * until the user enters an input of 0.
 */

public class LengthConverter {

   public static void main(String[] args) {

      double measurement;  // Numerical measurement, input by user.
      String units;        // The unit of measure for the input, also
                           //    specified by the user.

      double inches, feet, yards, miles;  // Measurement expressed in
                                          //   each possible unit of
                                          //   measure.

      TextIO.putln("Enter measurements in inches, feet, yards, or miles.");
      TextIO.putln("For example:  1 inch    17 feet    2.73 miles");
      TextIO.putln("You can use abbreviations:   in   ft  yd   mi");
      TextIO.putln("I will convert your input into the other units");
      TextIO.putln("of measure.");
      TextIO.putln();

      while (true) {

         /* Get the user's input, and convert units to lower case. */

         TextIO.put("Enter your measurement, or 0 to end:  ");
         measurement = TextIO.getDouble();
         if (measurement == 0)
            break;  // Terminate the while loop.
         units = TextIO.getlnWord();
         units = units.toLowerCase();

         /* Convert the input measurement to inches. */

         if (units.equals("inch") || units.equals("inches")
                                       || units.equals("in")) {
             inches = measurement;
         }
         else if (units.equals("foot") || units.equals("feet")
                                       || units.equals("ft")) {
             inches = measurement * 12;
         }
         else if (units.equals("yard") || units.equals("yards")
                                        || units.equals("yd")) {
             inches = measurement * 36;
         }
```

```
            else if (units.equals("mile") || units.equals("miles")
                                              || units.equals("mi")) {
                inches = measurement * 12 * 5280;
            }
            else {
                TextIO.putln("Sorry, but I don't understand \""
                                                    + units + "\".");
                continue;  // back to start of while loop
            }

            /* Convert measurement in inches to feet, yards, and miles. */

            feet = inches / 12;
            yards = inches / 36;
            miles = inches / (12*5280);

            /* Output measurement in terms of each unit of measure. */

            TextIO.putln();
            TextIO.putln("That's equivalent to:");
            TextIO.putf("%12.5g", inches);
            TextIO.putln(" inches");
            TextIO.putf("%12.5g", feet);
            TextIO.putln(" feet");
            TextIO.putf("%12.5g", yards);
            TextIO.putln(" yards");
            TextIO.putf("%12.5g", miles);
            TextIO.putln(" miles");
            TextIO.putln();

        } // end while

        TextIO.putln();
        TextIO.putln("OK!  Bye for now.");

    } // end main()

} // end class LengthConverter
```

(Note that this program uses formatted output with the "g" format specifier. In this program, we have no control over how large or how small the numbers might be. It could easily make sense for the user to enter very large or very small measurements. The "g" format will print a real number in exponential form if it is very large or very small, and in the usual decimal form otherwise. Remember that in the format specification `%12.5g`, the 5 is the total number of significant digits that are to be printed, so we will always get the same number of significant digits in the output, no matter what the size of the number. If we had used an "f" format specifier such as `%12.5f`, the output would be in decimal form with 5 digits after the decimal point. This would print the number 0.000000000745482 as `0.00000`, with no **significant** digits at all! With the "g" format specifier, the output would be `7.4549e-10`.)

3.5.4 The Empty Statement

As a final note in this section, I will mention one more type of statement in Java: the ***empty statement***. This is a statement that consists simply of a semicolon and which tells the computer

to do nothing. The existence of the empty statement makes the following legal, even though you would not ordinarily see a semicolon after a `}` :

```
if (x < 0) {
    x = -x;
};
```

The semicolon is legal after the `}`, but the computer considers it to be an empty statement, not part of the `if` statement. Occasionally, you might find yourself using the empty statement when what you mean is, in fact, "do nothing." For example, the rather contrived `if` statement

```
if ( done )
   ;  // Empty statement
else
   System.out.println( "Not done yet.");
```

does nothing when the **boolean** variable `done` is true, and prints out "Not done yet" when it is false. You can't just leave out the semicolon in this example, since Java syntax requires an actual statement between the `if` and the `else`. I prefer, though, to use an empty block, consisting of `{` and `}` with nothing between, for such cases.

Occasionally, stray empty statements can cause annoying, hard-to-find errors in a program. For example, the following program segment prints out "Hello" just **once**, not ten times:

```
for (int i = 0; i < 10; i++);
    System.out.println("Hello");
```

Why? Because the ";" at the end of the first line is a statement, and it is this statement that is executed ten times. The `System.out.println` statement is not really inside the `for` statement at all, so it is executed just once, after the `for` loop has completed.

3.6 The switch Statement

The second branching statement in Java is the `switch` statement, which is introduced in this section. The `switch` statement is used far less often than the `if` statement, but it is sometimes useful for expressing a certain type of multi-way branch.

3.6.1 The Basic switch Statement

A switch statement allows you to test the value of an expression and, depending on that value, to jump directly to some location within the switch statement. Only expressions of certain types can be used. The value of the expression can be one of the primitive integer types **int**, **short**, or **byte**. It can be the primitive **char** type. Or, as we will see later in this section, it can be an enumerated type. In Java 7, *Strings* are also allowed. In particular, the expression **cannot** be a real number, and prior to Java 7, it **cannot** be a *String*. The positions that you can jump to are marked with ***case labels*** that take the form: "case ⟨*constant*⟩:". This marks the position the computer jumps to when the expression evaluates to the given ⟨*constant*⟩. As the final case in a switch statement you can, optionally, use the label "default:", which provides a default jump point that is used when the value of the expression is not listed in any case label.

A `switch` statement, as it is most often used, has the form:

```
switch (⟨expression⟩) {
   case ⟨constant-1⟩:
      ⟨statements-1⟩
      break;
   case ⟨constant-2⟩:
      ⟨statements-2⟩
      break;
      .
      .   // (more cases)
      .
   case ⟨constant-N⟩:
      ⟨statements-N⟩
      break;
   default:  // optional default case
      ⟨statements-(N+1)⟩
} // end of switch statement
```

The `break` statements are technically optional. The effect of a `break` is to make the computer jump to the end of the switch statement. If you leave out the break statement, the computer will just forge ahead after completing one case and will execute the statements associated with the next case label. This is rarely what you want, but it is legal. (I will note here—although you won't understand it until you get to the next chapter—that inside a subroutine, the `break` statement is sometimes replaced by a `return` statement.)

Note that you can leave out one of the groups of statements entirely (including the `break`). You then have two case labels in a row, containing two different constants. This just means that the computer will jump to the same place and perform the same action for each of the two constants.

Here is an example of a switch statement. This is not a useful example, but it should be easy for you to follow. Note, by the way, that the constants in the case labels don't have to be in any particular order, as long as they are all different:

```
switch ( N ) {   // (Assume N is an integer variable.)
   case 1:
      System.out.println("The number is 1.");
      break;
   case 2:
   case 4:
   case 8:
      System.out.println("The number is 2, 4, or 8.");
      System.out.println("(That's a power of 2!)");
      break;
   case 3:
   case 6:
   case 9:
      System.out.println("The number is 3, 6, or 9.");
      System.out.println("(That's a multiple of 3!)");
      break;
   case 5:
      System.out.println("The number is 5.");
      break;
   default:
      System.out.println("The number is 7 or is outside the range 1 to 9.");
}
```

The switch statement is pretty primitive as control structures go, and it's easy to make mistakes when you use it. Java takes all its control structures directly from the older programming languages C and C++. The switch statement is certainly one place where the designers of Java should have introduced some improvements.

3.6.2 Menus and switch Statements

One application of `switch` statements is in processing menus. A menu is a list of options. The user selects one of the options. The computer has to respond to each possible choice in a different way. If the options are numbered 1, 2, ..., then the number of the chosen option can be used in a `switch` statement to select the proper response.

In a *TextIO*-based program, the menu can be presented as a numbered list of options, and the user can choose an option by typing in its number. Here is an example that could be used in a variation of the `LengthConverter` example from the previous section:

```
int optionNumber;   // Option number from menu, selected by user.
double measurement; // A numerical measurement, input by the user.
                    //    The unit of measurement depends on which
                    //    option the user has selected.
double inches;      // The same measurement, converted into inches.

/* Display menu and get user's selected option number. */

TextIO.putln("What unit of measurement does your input use?");
TextIO.putln();
TextIO.putln("         1.  inches");
TextIO.putln("         2.  feet");
TextIO.putln("         3.  yards");
TextIO.putln("         4.  miles");
TextIO.putln();
TextIO.putln("Enter the number of your choice: ");
optionNumber = TextIO.getlnInt();

/* Read user's measurement and convert to inches. */

switch ( optionNumber ) {
   case 1:
       TextIO.putln("Enter the number of inches: ");
       measurement = TextIO.getlnDouble();
       inches = measurement;
       break;
   case 2:
       TextIO.putln("Enter the number of feet: ");
       measurement = TextIO.getlnDouble();
       inches = measurement * 12;
       break;
   case 3:
       TextIO.putln("Enter the number of yards: ");
       measurement = TextIO.getlnDouble();
       inches = measurement * 36;
       break;
   case 4:
       TextIO.putln("Enter the number of miles: ");
       measurement = TextIO.getlnDouble();
```

```
        inches = measurement * 12 * 5280;
        break;
   default:
        TextIO.putln("Error!  Illegal option number!  I quit!");
        System.exit(1);
} // end switch

/* Now go on to convert inches to feet, yards, and miles... */
```

In Java 7, this example might be rewritten using a *String* in the `switch` statement:

```
String units;       // Unit of measurement, entered by user.
double measurement; // A numerical measurement, input by the user.
double inches;      // The same measurement, converted into inches.

/* Read the user's unit of measurement. */

TextIO.putln("What unit of measurement does your input use?");
TextIO.put("inches, feet, yards, or miles ?");
units = TextIO.getln().toLowerCase();

/* Read user's measurement and convert to inches. */

TextIO.put("Enter the number of " + units + ":  ");
measurement = TextIO.getlnDouble();

switch ( units ) {  // Requires Java 7 or higher!
   case "inches":
        inches = measurement;
        break;
   case "feet":
        inches = measurement * 12;
        break;
   case "yards":
        inches = measurement * 36;
        break;
   case "miles":
        inches = measurement * 12 * 5280;
        break;
   default:
        TextIO.putln("Wait a minute!  Illegal unit of measure!  I quit!");
        System.exit(1);
} // end switch
```

3.6.3 Enums in switch Statements

The type of the expression in a `switch` can be an enumerated type. In that case, the constants in the `case` labels must be values from the enumerated type. For example, if the type of the expression is the enumerated type *Season* defined by

```
enum Season { SPRING, SUMMER, FALL, WINTER }
```

then the constants in the case label must be chosen from among the values `Season.SPRING`, `Season.SUMMER`, `Season.FALL`, or `Season.WINTER`. However, there is another quirk in the syntax: when an enum constant is used in a `case` label, only the simple name, such as "`SPRING`" can be used, not the full name "`Season.SPRING`". Of course, the computer already knows that the value in the `case` label must belong to the enumerated type, since it can tell that from the

type of expression used, so there is really no need to specify the type name in the constant. As an example, suppose that `currentSeason` is a variable of type *Season*. Then we could have the `switch` statement:

```
switch ( currentSeason ) {
   case WINTER:    // ( NOT Season.WINTER ! )
      System.out.println("December, January, February");
      break;
   case SPRING:
      System.out.println("March, April, May");
      break;
   case SUMMER:
      System.out.println("June, July, August");
      break;
   case FALL:
      System.out.println("September, October, November");
      break;
}
```

3.6.4 Definite Assignment

As a somewhat more realistic example, the following `switch` statement makes a random choice among three possible alternatives. Recall that the value of the expression `(int)(3*Math.random())` is one of the integers 0, 1, or 2, selected at random with equal probability, so the `switch` statement below will assign one of the values `"Rock"`, `"Scissors"`, `"Paper"` to `computerMove`, with probability 1/3 for each case. Although the switch statement in this example is correct, this code segment as a whole illustrates a subtle syntax error that sometimes comes up:

```
String computerMove;
switch ( (int)(3*Math.random()) ) {
   case 0:
      computerMove = "Rock";
      break;
   case 1:
      computerMove = "Scissors";
      break;
   case 2:
      computerMove = "Paper";
      break;
}
System.out.println("Computer's move is " + computerMove);   // ERROR!
```

You probably haven't spotted the error, since it's not an error from a human point of view. The computer reports the last line to be an error, because the variable `computerMove` might not have been assigned a value. In Java, it is only legal to use the value of a variable if a value has already been ***definitely assigned*** to that variable. This means that the computer must be able to prove, just from looking at the code when the program is compiled, that the variable must have been assigned a value. Unfortunately, the computer only has a few simple rules that it can apply to make the determination. In this case, it sees a `switch` statement in which the type of expression is **int** and in which the cases that are covered are 0, 1, and 2. For other values of the expression, `computerMove` is never assigned a value. So, the computer thinks

`computerMove` might still be undefined after the `switch` statement. Now, in fact, this isn't true: 0, 1, and 2 are actually the only possible values of the expression `(int)(3*Math.random())`, but the computer isn't smart enough to figure that out. The easiest way to fix the problem is to replace the case label `case 2` with `default`. The computer can then see that a value is assigned to `computerMove` in all cases.

More generally, we say that a value has been definitely assigned to a variable at a given point in a program if every execution path leading from the declaration of the variable to that point in the code includes an assignment to the variable. This rule takes into account loops and `if` statements as well as `switch` statements. For example, the following two `if` statements both do the same thing as the `switch` statement given above, but only the one on the right definitely assigns a value to `computerMove`:

```
String computerMove;
int rand;
rand = (int)(3*Math.random());
if ( rand == 0 )
   computerMove = "Rock";
else if ( rand == 1 )
   computerMove = "Scissors";
else if ( rand == 2 )
   computerMove = "Paper";
```

```
String computerMove;
int rand;
rand = (int)(3*Math.random());
if ( rand == 0 )
   computerMove = "Rock";
else if ( rand == 1 )
   computerMove = "Scissors";
else
   computerMove = "Paper";
```

In the code on the left, the test "`if ( rand == 2 )`" in the final `else` clause is unnecessary because if `rand` is not 0 or 1, the only remaining possibility is that `rand == 2`. The computer, however, can't figure that out.

3.7 Introduction to Exceptions and try..catch

In addition to the control structures that determine the normal flow of control in a program, Java has a way to deal with "exceptional" cases that throw the flow of control off its normal track. When an error occurs during the execution of a program, the default behavior is to terminate the program and to print an error message. However, Java makes it possible to "catch" such errors and program a response different from simply letting the program crash. This is done with the ***try..catch*** statement. In this section, we will take a preliminary, incomplete look at using `try..catch` to handle errors. Error handling is a complex topic, which we will return to in Chapter 8.

3.7.1 Exceptions

The term ***exception*** is used to refer to the type of error that one might want to handle with a `try..catch`. An exception is an exception to the normal flow of control in the program. The term is used in preference to "error" because in some cases, an exception might not be considered to be an error at all. You can sometimes think of an exception as just another way to organize a program.

Exceptions in Java are represented as objects of type *Exception*. Actual exceptions are defined by subclasses of *Exception*. Different subclasses represent different types of exceptions. We will look at only two types of exception in this section: *NumberFormatException* and *IllegalArgumentException*.

A *NumberFormatException* can occur when an attempt is made to convert a string into a number. Such conversions are done by the functions `Integer.parseInt`

and `Double.parseDouble`. (See Subsection 2.5.7.) Consider the function call `Integer.parseInt(str)` where `str` is a variable of type *String*. If the value of `str` is the string `"42"`, then the function call will correctly convert the string into the **int** 42. However, if the value of `str` is, say, `"fred"`, the function call will fail because `"fred"` is not a legal string representation of an **int** value. In this case, an exception of type *NumberFormatException* occurs. If nothing is done to handle the exception, the program will crash.

An *IllegalArgumentException* can occur when an illegal value is passed as a parameter to a subroutine. For example, if a subroutine requires that a parameter be greater than or equal to zero, an *IllegalArgumentException* might occur when a negative value is passed to the subroutine. How to respond to the illegal value is up to the person who wrote the subroutine, so we can't simply say that every illegal parameter value will result in an *IllegalArgumentException*. However, it is a common response.

One case where an *IllegalArgumentException* can occur is in the `valueOf` function of an enumerated type. Recall from Subsection 2.3.3 that this function tries to convert a string into one of the values of the enumerated type. If the string that is passed as a parameter to `valueOf` is not the name of one of the enumerated type's values, then an *IllegalArgumentException* occurs. For example, given the enumerated type

```
enum Toss { HEADS, TAILS }
```

`Toss.valueOf("HEADS")` correctly returns the value `Toss.HEADS`, while `Toss.valueOf("FEET")` results in an *IllegalArgumentException*.

3.7.2 try..catch

When an exception occurs, we say that the exception is "thrown". For example, we say that `Integer.parseInt(str)` ***throws*** an exception of type *NumberFormatException* when the value of `str` is illegal. When an exception is thrown, it is possible to "catch" the exception and prevent it from crashing the program. This is done with a ***try..catch*** statement. In somewhat simplified form, the syntax for a `try..catch` is:

```
try {
   ⟨statements-1⟩
}
catch ( ⟨exception-class-name⟩  ⟨variable-name⟩ ) {
   ⟨statements-2⟩
}
```

The ⟨*exception-class-name*⟩ could be *NumberFormatException*, *IllegalArgumentException*, or some other exception class. When the computer executes this statement, it executes the statements in the `try` part. If no error occurs during the execution of ⟨*statements-1*⟩, then the computer just skips over the `catch` part and proceeds with the rest of the program. However, if an exception of type ⟨*exception-class-name*⟩ occurs during the execution of ⟨*statements-1*⟩, the computer immediately jumps to the `catch` part and executes ⟨*statements-2*⟩, skipping any remaining statements in ⟨*statements-1*⟩. During the execution of ⟨*statements-2*⟩, the ⟨*variable-name*⟩ represents the exception object, so that you can, for example, print it out. At the end of the `catch` part, the computer proceeds with the rest of the program; the exception has been caught and handled and does not crash the program. Note that only one type of exception is caught; if some other type of exception occurs during the execution of ⟨*statements-1*⟩, it will crash the program as usual.

By the way, note that the braces, `{` and `}`, are part of the syntax of the `try..catch` statement. They are required even if there is only one statement between the braces. This is different from the other statements we have seen, where the braces around a single statement are optional.

As an example, suppose that `str` is a variable of type *String* whose value might or might not represent a legal real number. Then we could say:

```
try {
   double x;
   x = Double.parseDouble(str);
   System.out.println( "The number is " + x );
}
catch ( NumberFormatException e ) {
   System.out.println( "Not a legal number." );
}
```

If an error is thrown by the call to `Double.parseDouble(str)`, then the output statement in the `try` part is skipped, and the statement in the `catch` part is executed.

It's not always a good idea to catch exceptions and continue with the program. Often that can just lead to an even bigger mess later on, and it might be better just to let the exception crash the program at the point where it occurs. However, sometimes it's possible to recover from an error. For example, suppose that we have the enumerated type

```
enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY }
```

and we want the user to input a value belonging to this type. `TextIO` does not know about this type, so we can only read the user's response as a string. The function `Day.valueOf` can be used to convert the user's response to a value of type *Day*. This will throw an exception of type *IllegalArgumentException* if the user's response is not the name of one of the values of type *Day*, but we can recover from the error easily enough by asking the user to enter another response. Here is a code segment that does this. (Converting the user's response to upper case will allow responses such as "Monday" or "monday" in addition to "MONDAY".)

```
Day  weekday;  // User's response as a value of type Day.
while ( true ) {
   String response;  // User's response as a String.
   System.out.print("Please enter a day of the week: ");
   response = TextIO.getln();
   response = response.toUpperCase();
   try {
      weekday = Day.valueOf(response);
      break;
   }
   catch ( IllegalArgumentException e ) {
      System.out.println( response + " is not the name of a day of the week." );
   }
}
// At this point, a legal value has definitely been assigned to weekday.
```

The `break` statement will be reached only if the user's response is acceptable, and so the loop will end only when a legal value has been assigned to `weekday`.

3.7.3 Exceptions in TextIO

When `TextIO` reads a numeric value from the user, it makes sure that the user's response is legal, using a technique similar to the `while` loop and `try..catch` in the previous example. However, `TextIO` can read data from other sources besides the user. (See Subsection 2.4.5.) When it is reading from a file, there is no reasonable way for `TextIO` to recover from an illegal value in the input, so it responds by throwing an exception. To keep things simple, `TextIO` only throws exceptions of type *IllegalArgumentException*, no matter what type of error it encounters. For example, an exception will occur if an attempt is made to read from a file after all the data in the file has already been read. In `TextIO`, the exception is of type *IllegalArgumentException*. If you have a better response to file errors than to let the program crash, you can use a `try..catch` to catch exceptions of type *IllegalArgumentException*.

For example, suppose that a file contains nothing but real numbers, and we want a program that will read the numbers and find their sum and their average. Since it is unknown how many numbers are in the file, there is the question of when to stop reading. One approach is simply to try to keep reading indefinitely. When the end of the file is reached, an exception occurs. This exception is not really an error—it's just a way of detecting the end of the data, so we can catch the exception and finish up the program. We can read the data in a `while (true)` loop and break out of the loop when an exception occurs. This is an example of the somewhat unusual technique of using an exception as part of the expected flow of control in a program.

To read from the file, we need to know the file's name. To make the program more general, we can let the user enter the file name, instead of hard-coding a fixed file name in the program. However, it is possible that the user will enter the name of a file that does not exist. When we use `TextIO.readfile` to open a file that does not exist, an exception of type *IllegalArgumentException* occurs. We can catch this exception and ask the user to enter a different file name. Here is a complete program that uses all these ideas:

```
/**
 * This program reads numbers from a file.  It computes the sum and
 * the average of the numbers that it reads.  The file should contain
 * nothing but numbers of type double; if this is not the case, the
 * output will be the sum and average of however many numbers were
 * successfully read from the file.  The name of the file will be
 * input by the user.
 */

public class ReadNumbersFromFile {

   public static void main(String[] args) {

      while (true) {
         String fileName;  // The name of the file, to be input by the user.
         TextIO.put("Enter the name of the file: ");
         fileName = TextIO.getln();
         try {
            TextIO.readFile( fileName );  // Try to open the file for input.
            break;  // If that succeeds, break out of the loop.
         }
         catch ( IllegalArgumentException e ) {
            TextIO.putln("Can't read from the file \"" + fileName + "\".");
            TextIO.putln("Please try again.\n");
         }
```

```
        }

        // At this point, TextIO is reading from the file.

        double number;  // A number read from the data file.
        double sum;     // The sum of all the numbers read so far.
        int count;      // The number of numbers that were read.

        sum = 0;
        count = 0;

        try {
           while (true) { // Loop ends when an exception occurs.
               number = TextIO.getDouble();
               count++;  // This is skipped when the exception occurs
               sum += number;
           }
        }
        catch ( IllegalArgumentException e ) {
           // We expect this to occur when the end-of-file is encountered.
           // We don't consider this to be an error, so there is nothing to do
           // in this catch clause.  Just proceed with the rest of the program.
        }

        // At this point, we've read the entire file.

        TextIO.putln();
        TextIO.putln("Number of data values read: " + count);
        TextIO.putln("The sum of the data values: " + sum);
        if ( count == 0 )
           TextIO.putln("Can't compute an average of 0 values.");
        else
           TextIO.putln("The average of the values:  " + (sum/count));

     }

  }
```

3.8 Introduction to GUI Programming

For the past two chapters, you've been learning the sort of programming that is done inside a single subroutine. In the rest of the text, we'll be more concerned with the larger scale structure of programs, but the material that you've already learned will be an important foundation for everything to come.

In this section, before moving on to programming-in-the-large, we'll take a look at how programming-in-the-small can be used in other contexts besides text-based, command-line-style programs. We'll do this by taking a short, introductory look at applets and graphical programming. The point here is not so much to understand GUI programming as it is to illustrate that a knowledge of programming-in-the-small applies to writing the guts of any subroutine, not just `main()`.

An ***applet*** is a Java program that runs on a Web page. An applet is not a stand-alone application, and it does not have a `main()` routine. In fact, an applet is an **object** rather than a **class**. When Java first appeared on the scene, applets were one of its major appeals. Since then, they have become much less important, although they can still be very useful. When

we study GUI programming in Chapter 6, we will concentrate on stand-alone GUI programs rather than on applets, but applets are a good place to start for our first look at the subject.

When an applet is placed on a Web page, it is assigned a rectangular area on the page. It is the job of the applet to draw the contents of that rectangle. When the region needs to be drawn, the Web page calls a subroutine in the applet to do so. This is not so different from what happens with stand-alone programs. When such a program needs to be run, the system calls the `main()` routine of the program. Similarly, when an applet needs to be drawn, the Web page calls a subroutine in the applet. The programmer specifies what happens when this routine is called by filling in the body of the routine. Programming in the small! Applets can do other things besides draw themselves, such as responding when the user clicks the mouse on the applet. Each of the applet's behaviors is defined by a subroutine. The programmer specifies how the applet behaves by filling in the bodies of the appropriate subroutines.

To define an applet, you need a class that is a subclass of the built-in class named *Applet*. To avoid some technicalities in this section as well as to make things a little more interesting, we will not work with the *Applet* class directly. Instead, we will work with a class that I wrote named *AnimationBase*, which is itself a subclass of *Applet*. *AnimationBase* makes it easy to write simple animations. A ***computer animation*** is really just a sequence of still images, which are called the ***frames*** of the animation. The computer displays the images one after the other. Each image differs a bit from the preceding image in the sequence. If the differences are not too big and if the sequence is displayed quickly enough, the eye is tricked into perceiving continuous motion. To create the animation, you just have to say how to draw each individual frame. When using *AnimationBase*, you do that by filling in the inside of a subroutine named `drawFrame()`. More specifically, to create an animation using *AnimationBase*, you have write a class of the form:

```
import java.awt.*;

public class ⟨name-of-class⟩ extends AnimationBase {

    public void drawFrame(Graphics g) {
       ⟨statements⟩
    }

}
```

where ⟨*name-of-class*⟩ is an identifier that names the class, and the ⟨*statements*⟩ are the code that actually draws the content of one of the frames of the animation. This looks similar to the definition of a stand-alone program, but there are a few things here that need to be explained, starting with the first line.

When you write a program, there are certain built-in classes that are available for you to use. These built-in classes include *System* and *Math*. If you want to use one of these classes, you don't have to do anything special. You just go ahead and use it. But Java also has a large number of standard classes that are there if you want them but that are not automatically available to your program. (There are just too many of them.) If you want to use these classes in your program, you have to ask for them first. The standard classes are grouped into so-called "packages." One of these packages is called "java.awt". The directive "import java.awt.*;" makes all the classes from the package java.awt available for use in your program. The `java.awt` package contains classes related to graphical user interface programming, including a class called `Graphics`. The `Graphics` class is referred to in the `drawFrame()` routine above and will be used for drawing the frame.

The definition of the class above says that the class "extends `AnimationBase`." The *AnimationBase* class includes all the basic properties and behaviors of applet objects (since it is a subclass of *Applet*). It also defines the basic properties and behaviors of animations—it "extends" class *Applet* by adding in this extra stuff. When you extend *AnimationBase*, you inherit all these properties and behaviors, and you can add even more stuff, in particular the drawing commands that you want to use to create your animation.

(One more thing needs to be mentioned—and this is a point where Java's syntax gets unfortunately confusing. You can skip this explanation until Chapter 5 if you want. Applets are objects, not classes. Instead of being static members of a class, the subroutines that define the applet's behavior are part of the applet object. We say that they are "non-static" subroutines. Of course, objects are related to classes because every object is described by a class. Now here is the part that can get confusing: Even though a non-static subroutine is not actually part of a class (in the sense of being part of the behavior of the class itself), it is nevertheless defined in a class (in the sense that the Java code that defines the subroutine is part of the Java code that defines the class). Many objects can be described by the same class. Each object has its own non-static subroutine. But the common definition of those subroutines—the actual Java source code—is physically part of the class that **describes** all the objects. To put it briefly: static subroutines in a class definition say what the class does; non-static subroutines say what all the objects described by the class do. The `drawFrame()` routine is an example of a non-static subroutine. A stand-alone program's `main()` routine is an example of a static subroutine. The distinction doesn't really matter too much at this point: When working with stand-alone programs, mark everything with the reserved word, "`static`"; leave it out when working with applets. However, the distinction between static and non-static will become more important later in the course.)

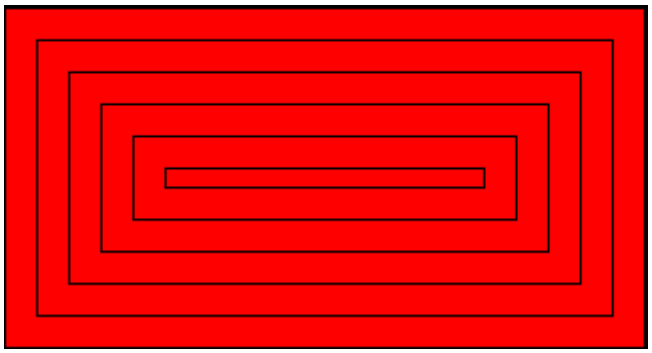
* * *

Let's write an applet based on *AnimationBase*. In order to draw the content, we'll need to know some basic subroutines that are already available for drawing, just as in writing text-oriented programs we need to know what subroutines are available for reading and writing text. In Java, the built-in drawing subroutines are found in objects of the class *Graphics*, one of the classes in the `java.awt` package. In our applet's `drawFrame()` routine, we can use the *Graphics* object `g` for drawing. (This object is provided as a parameter to the `drawFrame()` routine when that routine is called.) *Graphics* objects contain many subroutines. I'll mention just three of them here. You'll encounter more of them in Chapter 6.

- `g.setColor(c)`, is called to set the color that is used for drawing. The parameter, `c` is an object belonging to a class named *Color*, another one of the classes in the java.awt package. About a dozen standard colors are available as static member variables in the *Color* class. These standard colors include `Color.BLACK`, `Color.WHITE`, `Color.RED`, `Color.GREEN`, and `Color.BLUE`. For example, if you want to draw in red, you would say "`g.setColor(Color.RED);`". The specified color is used for all subsequent drawing operations up until the next time `setColor()` is called.
- `g.drawRect(x,y,w,h)` draws the outline of a rectangle. The parameters `x`, `y`, `w`, and `h` must be integers or integer-valued expressions. This subroutine draws the outline of the rectangle whose top-left corner is `x` pixels from the left edge of the applet and `y` pixels down from the top of the applet. The width of the rectangle is `w` pixels, and the height is `h` pixels. The color that is used is black, unless a different color has been set by calling `setColor()`.

- `g.fillRect(x,y,w,h)` is similar to `drawRect` except that it fills in the inside of the rectangle instead of just drawing an outline.

This is enough information to write an applet that will draw the following image on a Web page:



Although the applet is defined as an animation, you don't see any movement because all the frames that are drawn are identical! This is rather silly, and we will fix it in the next example. But for now, we are just interested in seeing how to use drawing routines to draw a picture.

The applet first fills its entire rectangular area with red. Then it changes the drawing color to black and draws a sequence of rectangles, where each rectangle is nested inside the previous one. The rectangles can be drawn with a `while` loop, which draws the rectangles starting from the outside and moving in. Each time through the loop, the rectangle that is drawn is smaller than the previous one and is moved down and over a bit. We'll need variables to hold the width and height of the rectangle and a variable to record how far the top-left corner of the rectangle is inset from the edges of the applet. The while loop ends when the rectangle shrinks to nothing. In general outline, the algorithm for drawing the applet is

```
Set the drawing color to red  (using the g.setColor subroutine)
Fill in the entire applet (using the g.fillRect subroutine)
Set the drawing color to black
Set the top-left corner inset to be 0
Set the rectangle width and height to be as big as the applet
while the width and height are greater than zero:
    draw a rectangle (using the g.drawRect subroutine)
    increase the inset
    decrease the width and the height
```

In my applet, each rectangle is 15 pixels away from the rectangle that surrounds it, so the `inset` is increased by 15 each time through the `while` loop. The rectangle shrinks by 15 pixels on the left **and** by 15 pixels on the right, so the width of the rectangle shrinks by 30 each time through the loop. The height also shrinks by 30 pixels each time through the loop.

It is not hard to code this algorithm into Java and use it to define the `drawFrame()` method of the applet. I've assumed that the applet has a height of 160 pixels and a width of 300 pixels. The size is actually set in the source code of the Web page where the applet appears. In order for an applet to appear on a page, the source code for the page must include a command that specifies which applet to run and how big it should be. (We'll see how to do that later; see Exercise 3.6 and Section 6.2.) It's not a great idea to assume that we know how big the applet is going to be, as I do here; I'll address that issue before the end of this section. But for now, here is the source code for the applet:

```
import java.awt.*;

public class StaticRects extends AnimationBase {

     public void drawFrame(Graphics g) {

         // Draw set of nested black rectangles on a red background.
         // Each nested rectangle is separated by 15 pixels on all sides
         // from the rectangle that encloses it.  The applet is
         // assumed to be 300 pixels wide and 160 pixels high.

       int inset;    // Gap between borders of applet and one of the rectangles.

       int rectWidth, rectHeight;   // The size of one of the rectangles.

       g.setColor(Color.red);
       g.fillRect(0,0,300,160);  // Fill the entire applet with red.

       g.setColor(Color.black);  // Draw the rectangles in black.

       inset = 0;

       rectWidth = 299;    // Set size of the first rect to size of applet
       rectHeight = 159;

       while (rectWidth >= 0 && rectHeight >= 0) {
          g.drawRect(inset, inset, rectWidth, rectHeight);
          inset += 15;       // rects are 15 pixels apart
          rectWidth -= 30;   // width decreases by 15 pixels on left and 15 on right
          rectHeight -= 30;  // height decreases by 15 pixels on top and 15 on bottom
       }

    }  // end paint()

}  // end class StaticRects
```

(You might wonder why the initial `rectWidth` is set to 299, instead of to 300, since the width of the applet is 300 pixels. It's because rectangles are drawn as if with a pen whose nib hangs below and to the right of the point where the pen is placed. If you run the pen exactly along the right edge of the applet, the line it draws is actually outside the applet and therefore is not seen. So instead, we run the pen along a line one pixel to the left of the edge of the applet. The same reasoning applies to `rectHeight`. Careful graphics programming demands attention to details like these.)

* * *

When you write an animation applet, you get to build on *AnimationBase* which in turn builds on the work of the people who wrote the *Applet* class. The *AnimationBase* class provides a framework on which you can hang your own work. Any programmer can create additional frameworks that can be used by other programmers as a basis for writing specific types of applets or stand-alone programs. This makes it possible for other programmers to build on their work even without understanding in detail what goes on "inside" the code that they wrote. This type of thing is the key to building complex systems!

Let's continue our example by animating the rectangles in our applet. You can see the animation in action at the bottom of the on-line version of this section.

In the animation, rectangles shrink continually towards the center of the applet, while new rectangles appear at the edge. The perpetual motion is, of course, an illusion. If you think about it, you'll see that the animation loops through the same set of images over and over.

In each image, there is a gap between the borders of the applet and the outermost rectangle. This gap gets wider and wider until a new rectangle appears at the border. Only it's not a new rectangle. You are seeing a picture that is identical to the first picture that was drawn. What has really happened is that the animation has started over again with the first image in the sequence.

In order to create motion in the animation, `drawFrame()` will have to draw a different picture each time it is called. How can it do that? The picture that should be drawn will depend on the ***frame number***, that is, how many frames have been drawn so far. To find out the current frame number, we can use a function that is built into the *AnimationBase* class. This class provides the function named `getFrameNumber()` that you can call to find out the current frame number. This function returns the current frame number as an integer value. If the value returned is 0, you are supposed to draw the first frame; if the value is 1, you are supposed to draw the second frame, and so on. Depending on the frame number, the `drawFrame()` method will draw different pictures.

In the animation that we are writing, the thing that differs from one frame to another is the distance between the edges of the applet and the outermost rectangle. Since the rectangles are 15 pixels apart, this distance increases from 0 to 14 and then jumps back to 0 when a "new" rectangle appears. The appropriate value can be computed very simply from the frame number, with the statement "`inset = getFrameNumber() % 15;`". The value of the expression `getFrameNumber() % 15` is always between 0 and 14. When the frame number reaches 15 or any multiple of 15, the value of `getFrameNumber() % 15` jumps back to 0.

Drawing one frame in the sample animated applet is very similar to drawing the single image of the original *StaticRects* applet. We only have to make a few changes to the `drawFrame()` method. I've chosen to make one additional improvement: The *StaticRects* applet assumes that the applet is exactly 300 by 160 pixels. The new version, *MovingRects*, will work for any applet size. To implement this, the `drawFrame()` routine has to know how big the applet is. There are two functions that can be called to get this information. The function `getWidth()` returns an integer value representing the width of the applet, and the function `getHeight()` returns the height. These functions are inherited from the *Applet* class. The width and height, together with the frame number, are used to compute the size of the first rectangle that is drawn. Here is the complete source code:

```
import java.awt.*;

public class MovingRects extends AnimationBase {

  public void init() {
        // The init() method is called when the applet is first
        // created and can be used to initialize the applet.
        // Here, it is used to change the number of milliseconds
        // per frame from the default 100 to 30.  The faster
        // animation looks better.
     setMillisecondsPerFrame(30);
  }

  public void drawFrame(Graphics g) {

         // Draw one frame in the animation by filling in the background
         // with a solid red and then drawing a set of nested black
         // rectangles.  The frame number tells how much the first
         // rectangle is to be inset from the borders of the applet.
```

```
        int width;    // Width of the applet, in pixels.
        int height;   // Height of the applet, in pixels.

        int inset;    // Gap between borders of applet and a rectangle.
                      //    The inset for the outermost rectangle goes from 0 to
                      //    14 then back to 0, and so on, as the frameNumber varies.

        int rectWidth, rectHeight;   // the size of one of the rectangles

        width = getWidth();              // find out the size of the drawing area
        height = getHeight();

        g.setColor(Color.red);           // fill the frame with red
        g.fillRect(0,0,width,height);

        g.setColor(Color.black);         // switch color to black

        inset = getFrameNumber() % 15;   // get the inset for the outermost rect

        rectWidth = width - 2*inset - 1;    // set size of the outermost rect
        rectHeight = height - 2*inset - 1;

        while (rectWidth >= 0 && rectHeight >= 0) {
           g.drawRect(inset,inset,rectWidth,rectHeight);
           inset += 15;       // rects are 15 pixels apart
           rectWidth -= 30;   // width decreases by 15 pixels on left and 15 on right
           rectHeight -= 30;  // height decreases by 15 pixels on top and 15 on bottom
        }

     }  // end drawFrame()

  }  // end class MovingRects
```

The main point here is that by building on an existing framework, you can do interesting things using the type of local, inside-a-subroutine programming that was covered in Chapter 2 and Chapter 3. As you learn more about programming and more about Java, you'll be able to do more on your own—but no matter how much you learn, you'll always be dependent on other people's work to some extent.

Exercises for Chapter 3

1. How many times do you have to roll a pair of dice before they come up snake eyes? You could do the experiment by rolling the dice by hand. Write a computer program that simulates the experiment. The program should report the number of rolls that it makes before the dice come up snake eyes. (Note: "Snake eyes" means that both dice show a value of 1.) Exercise 2.2 explained how to simulate rolling a pair of dice.

2. Which integer between 1 and 10000 has the largest number of divisors, and how many divisors does it have? Write a program to find the answers and print out the results. It is possible that several integers in this range have the same, maximum number of divisors. Your program only has to print out one of them. An example in Subsection 3.4.2 discussed divisors. The source code for that example is *CountDivisors.java*.

You might need some hints about how to find a maximum value. The basic idea is to go through all the integers, keeping track of the largest number of divisors that you've seen *so far*. Also, keep track of the integer that had that number of divisors.

3. Write a program that will evaluate simple expressions such as 17 + 3 and 3.14159 * 4.7. The expressions are to be typed in by the user. The input always consist of a number, followed by an operator, followed by another number. The operators that are allowed are +, -, *, and /. You can read the numbers with `TextIO.getDouble()` and the operator with `TextIO.getChar()`. Your program should read an expression, print its value, read another expression, print its value, and so on. The program should end when the user enters 0 as the first number on the line.

4. Write a program that reads one line of input text and breaks it up into words. The words should be output one per line. A word is defined to be a sequence of letters. Any characters in the input that are not letters should be discarded. For example, if the user inputs the line

```
He said, "That's not a good idea."
```

then the output of the program should be

```
He
said
That
s
not
a
good
idea
```

An improved version of the program would list "that's" as a single word. An apostrophe can be considered to be part of a word if there is a letter on each side of the apostrophe.

To test whether a character is a letter, you might use `(ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z')`. However, this only works in English and similar languages. A better choice is to call the standard function `Character.isLetter(ch)`, which returns a boolean value of `true` if `ch` is a letter and `false` if it is not. This works for any Unicode character.

5. Suppose that a file contains information about sales figures for a company in various cities. Each line of the file contains a city name, followed by a colon (:) followed by the data for that city. The data is a number of type **double.** However, for some cities, no data was available. In these lines, the data is replaced by a comment explaining why the data is missing. For example, several lines from the file might look like:

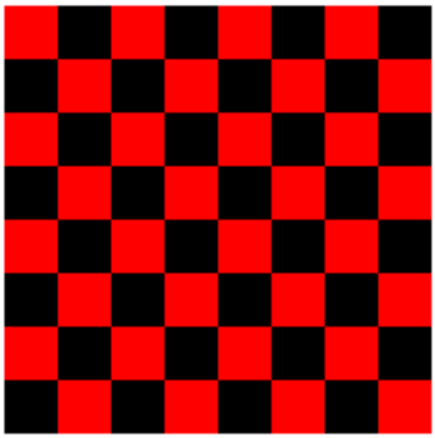
```
San Francisco:  19887.32
Chicago:  no report received
New York: 298734.12
```

Write a program that will compute and print the total sales from all the cities together. The program should also report the number of cities for which data was not available. The name of the file is "sales.dat".

To complete this program, you'll need one fact about file input with *TextIO* that was not covered in Subsection 2.4.5. Since you don't know in advance how many lines there are in the file, you need a way to tell when you have gotten to the end of the file. When *TextIO* is reading from a file, the function `TextIO.eof()` can be used to test for ***end of file***. This **boolean**-valued function returns `true` if the file has been entirely read and returns `false` if there is more data to read in the file. This means that you can read the lines of the file in a loop `while (TextIO.eof() == false)...`. The loop will end when all the lines of the file have been read.

Suggestion: For each line, read and ignore characters up to the colon. Then read the rest of the line into a variable of type *String*. Try to convert the string into a number, and use `try..catch` to test whether the conversion succeeds.

6. Write an applet that draws a checkerboard. Write your solution as a subclass of *AnimationBase*, even though all the frames that it draws will be the same. Assume that the size of the applet is 160 by 160 pixels. Each square in the checkerboard is 20 by 20 pixels. The checkerboard contains 8 rows of squares and 8 columns. The squares are red and black. Here is a tricky way to determine whether a given square should be red or black: If the row number and the column number are either both even or both odd, then the square is red. Otherwise, it is black. Note that a square is just a rectangle in which the height is equal to the width, so you can use the subroutine `g.fillRect()` to draw the squares. Here is an image of the checkerboard:



(To run an applet, you need a Web page to display it. A very simple page will do. Assume that your applet class is called `Checkerboard`, so that when you compile it you get a class file named `Checkerboard.class` Make a file that contains only the lines:

```
<applet code="Checkerboard.class" width=160 height=160>
</applet>
```

Call this file `Checkerboard.html`. This is the source code for a simple Web page that shows nothing but your applet. The compiled class file, `Checkerboard.class`, must be in the same directory with the Web-page file, `Checkerboard.html`. Furthermore, since your program depends on the non-standard class *AnimationBase*, you also have to make that class available to your program. To do this, you should compile the source code, *AnimationBase.java*. You can find a copy on the Source Code page of the on-line version of this book. The result will be **two** class files, `AnimationBase.class` and `AnimationBase$1.class`. Place **both** of these class files in the same directory, together with `Checkerboard.html` and `Checkerboard.class`. Now, to run the applet, simply open `Checkerboard.html` in a web browser. Alternatively, on the command line, you can use the command

```
appletviewer Checkerboard.html
```

The `appletviewer` command, like `java` and `javac` is part of a standard installation of the JDK.

If you are using the Eclipse Integrated Development Environment, you should add *AnimationBase.java* to the project where you want to write `Checkerboard.java`. You can then simply right-click the name of the source code file in the Package Explorer. In the pop-up menu, go to "Run As" then to "Java Applet". This will open the window in which the applet appears. The default size for the window is bigger than 160-by-160, so the drawing of the checkerboard will not fill the entire window.)

7. Write an animation applet that shows a checkerboard pattern in which the even numbered rows slide to the left while the odd numbered rows slide to the right. You can assume that the applet is 160 by 160 pixels. Each row can be offset towards the left or right from its usual position by the amount `getFrameNumber() % 40`. Hints: Anything you draw outside the boundaries of the applet will be invisible, so you can draw more than 8 squares in a row. You can use negative values of x in `g.fillRect(x,y,w,h)`. (Before trying to do this exercise, it would be a good idea to look at a working applet, which can be found in the on-line version of this book.)

As with Exercise 3.6, you can write your class as a subclass of *AnimationBase*. Compile and run the program in the same way, as described in that exercise. Assuming that the name of your class is `SlidingCheckerboard`, then the source file for the Web page this time should contain the lines:

```
<applet code="SlidingCheckerboard.class" width=160 height=160>
</applet>
```

Quiz on Chapter 3

1. What is an *algorithm*?

2. Explain briefly what is meant by "pseudocode" and how is it useful in the development of algorithms.

3. What is a *block statement?* How are block statements used in Java programs?

4. What is the main difference between a `while` loop and a `do..while` loop?

5. What does it mean to *prime* a loop?

6. Explain what is meant by an *animation* and how a computer displays an animation.

7. Write a `for` loop that will print out all the multiples of 3 from 3 to 36, that is: 3 6 9 12 15 18 21 24 27 30 33 36.

8. Fill in the following `main()` routine so that it will ask the user to enter an integer, read the user's response, and tell the user whether the number entered is even or odd. (You can use `TextIO.getInt()` to read the integer. Recall that an integer `n` is even if `n % 2 == 0`.)

```
public static void main(String[] args) {

        // Fill in the body of this subroutine!

}
```

9. Suppose that `s1` and `s2` are variables of type *String*, whose values are expected to be string representations of values of type **int**. Write a code segment that will compute and print the integer sum of those values, or will print an error message if the values cannot successfully be converted into integers. (Use a `try..catch` statement.)

10. Show the exact output that would be produced by the following `main()` routine:

```
public static void main(String[] args) {
    int N;
    N = 1;
    while (N <= 32) {
       N = 2 * N;
       System.out.println(N);
    }
}
```

11. Show the exact output produced by the following `main()` routine:

```
public static void main(String[] args) {
   int x,y;
   x = 5;
   y = 1;
   while (x > 0) {
      x = x - 1;
      y = y * x;
      System.out.println(y);
   }
}
```

12. What output is produced by the following program segment? **Why?** (Recall that `name.charAt(i)` is the i-th character in the string, `name`.)

```
String name;
int i;
boolean startWord;

name = "Richard M. Nixon";
startWord = true;
for (i = 0; i < name.length(); i++) {
   if (startWord)
      System.out.println(name.charAt(i));
   if (name.charAt(i) == ' ')
      startWord = true;
   else
      startWord = false;
}
```

Chapter 4

Programming in the Large I: Subroutines

One way to break up a complex program into manageable pieces is to use ***subroutines***. A subroutine consists of the instructions for carrying out a certain task, grouped together and given a name. Elsewhere in the program, that name can be used as a stand-in for the whole set of instructions. As a computer executes a program, whenever it encounters a subroutine name, it executes all the instructions necessary to carry out the task associated with that subroutine.

Subroutines can be used over and over, at different places in the program. A subroutine can even be used inside another subroutine. This allows you to write simple subroutines and then use them to help write more complex subroutines, which can then be used in turn in other subroutines. In this way, very complex programs can be built up step-by-step, where each step in the construction is reasonably simple.

As mentioned in Section 3.8, subroutines in Java can be either static or non-static. This chapter covers static subroutines only. Non-static subroutines, which are used in true object-oriented programming, will be covered in the next chapter.

4.1 Black Boxes

A subroutine consists of instructions for performing some task, chunked together and given a name. "Chunking" allows you to deal with a potentially very complicated task as a single concept. Instead of worrying about the many, many steps that the computer might have to go though to perform that task, you just need to remember the name of the subroutine. Whenever you want your program to perform the task, you just call the subroutine. Subroutines are a major tool for dealing with complexity.

A subroutine is sometimes said to be a "black box" because you can't see what's "inside" it (or, to be more precise, you usually don't **want** to see inside it, because then you would have to deal with all the complexity that the subroutine is meant to hide). Of course, a black box that has no way of interacting with the rest of the world would be pretty useless. A black box needs some kind of ***interface*** with the rest of the world, which allows some interaction between what's inside the box and what's outside. A physical black box might have buttons on the outside that you can push, dials that you can set, and slots that can be used for passing information back and forth. Since we are trying to hide complexity, not create it, we have the first rule of black boxes:

> **The interface of a black box should be fairly straightforward, well-defined, and easy to understand.**

Are there any examples of black boxes in the real world? Yes; in fact, you are surrounded by them. Your television, your car, your mobile phone, your refrigerator.... You can turn your television on and off, change channels, and set the volume by using elements of the television's interface—dials, remote control, don't forget to plug in the power—without understanding anything about how the thing actually works. The same goes for a mobile phone, although the interface in that case is a lot more complicated.

Now, a black box does have an inside—the code in a subroutine that actually performs the task, all the electronics inside your television set. The inside of a black box is called its ***implementation***. The second rule of black boxes is that:

> **To use a black box, you shouldn't need to know anything about its implementation; all you need to know is its interface.**

In fact, it should be possible to **change** the implementation, as long as the behavior of the box, as seen from the outside, remains unchanged. For example, when the insides of TV sets went from using vacuum tubes to using transistors, the users of the sets didn't even need to know about it—or even know what it means. Similarly, it should be possible to rewrite the inside of a subroutine, to use more efficient code, for example, without affecting the programs that use that subroutine.

Of course, to have a black box, someone must have designed and built the implementation in the first place. The black box idea works to the advantage of the implementor as well as the user of the black box. After all, the black box might be used in an unlimited number of different situations. The implementor of the black box doesn't need to know about any of that. The implementor just needs to make sure that the box performs its assigned task and interfaces correctly with the rest of the world. This is the third rule of black boxes:

> **The implementor of a black box should not need to know anything about the larger systems in which the box will be used.**

In a way, a black box divides the world into two parts: the inside (implementation) and the outside. The interface is at the boundary, connecting those two parts.

* * *

By the way, you should **not** think of an interface as just the physical connection between the box and the rest of the world. The interface also includes a ***specification*** of what the box does and how it can be controlled by using the elements of the physical interface. It's not enough to say that a TV set has a power switch; you need to specify that the power switch is used to turn the TV on and off!

To put this in computer science terms, the interface of a subroutine has a semantic as well as a syntactic component. The syntactic part of the interface tells you just what you have to type in order to call the subroutine. The semantic component specifies exactly what task the subroutine will accomplish. To write a legal program, you need to know the syntactic specification of the subroutine. To understand the purpose of the subroutine and to use it effectively, you need to know the subroutine's semantic specification. I will refer to both parts of the interface—syntactic and semantic—collectively as the ***contract*** of the subroutine.

The contract of a subroutine says, essentially, "Here is what you have to do to use me, and here is what I will do for you, guaranteed." When you write a subroutine, the comments that you write for the subroutine should make the contract very clear. (I should admit that in practice, subroutines' contracts are often inadequately specified, much to the regret and annoyance of the programmers who have to use them.)

For the rest of this chapter, I turn from general ideas about black boxes and subroutines in general to the specifics of writing and using subroutines in Java. But keep the general ideas and principles in mind. They are the reasons that subroutines exist in the first place, and they are your guidelines for using them. This should be especially clear in Section 4.6, where I will discuss subroutines as a tool in program development.

* * *

You should keep in mind that subroutines are not the only example of black boxes in programming. For example, a class is also a black box. We'll see that a class can have a "public" part, representing its interface, and a "private" part that is entirely inside its hidden implementation. All the principles of black boxes apply to classes as well as to subroutines.

4.2 Static Subroutines and Static Variables

Every subroutine in Java must be defined inside some class. This makes Java rather unusual among programming languages, since most languages allow free-floating, independent subroutines. One purpose of a class is to group together related subroutines and variables. Perhaps the designers of Java felt that everything must be related to something. As a less philosophical motivation, Java's designers wanted to place firm controls on the ways things are named, since a Java program potentially has access to a huge number of subroutines created by many different programmers. The fact that those subroutines are grouped into named classes (and classes are grouped into named "packages") helps control the confusion that might result from so many different names.

A subroutine that is a member of a class is often called a ***method***, and "method" is the term that most people prefer for subroutines in Java. I will start using the term "method" occasionally; however, I will continue to prefer the more general term "subroutine" in this chapter, at least for static subroutines. This chapter will deal with static subroutines almost exclusively. We'll turn to non-static methods and object-oriented programming in the next chapter.

4.2.1 Subroutine Definitions

A subroutine definition in Java takes the form:

```
⟨modifiers⟩ ⟨return-type⟩ ⟨subroutine-name⟩ ( ⟨parameter-list⟩ ) {
    ⟨statements⟩
}
```

It will take us a while—most of the chapter—to get through what all this means in detail. Of course, you've already seen examples of subroutines in previous chapters, such as the `main()` routine of a program and the `drawFrame()` routine of the animation applets in Section 3.8. So you are familiar with the general format.

The ⟨*statements*⟩ between the braces, `{` and `}`, in a subroutine definition make up the ***body*** of the subroutine. These statements are the inside, or implementation part, of the "black box",

as discussed in the previous section. They are the instructions that the computer executes when the method is called. Subroutines can contain any of the statements discussed in Chapter 2 and Chapter 3.

The ⟨*modifiers*⟩ that can occur at the beginning of a subroutine definition are words that set certain characteristics of the subroutine, such as whether it is static or not. The modifiers that you've seen so far are "`static`" and "`public`". There are only about a half-dozen possible modifiers altogether.

If the subroutine is a function, whose job is to compute some value, then the ⟨*return-type*⟩ is used to specify the type of value that is returned by the function. We'll be looking at functions and return types in some detail in Section 4.4. If the subroutine is not a function, then the ⟨*return-type*⟩ is replaced by the special value `void`, which indicates that no value is returned. The term "void" is meant to indicate that the return value is empty or non-existent.

Finally, we come to the ⟨*parameter-list*⟩ of the method. Parameters are part of the interface of a subroutine. They represent information that is passed into the subroutine from outside, to be used by the subroutine's internal computations. For a concrete example, imagine a class named *Television* that includes a method named `changeChannel()`. The immediate question is: What channel should it change to? A parameter can be used to answer this question. Since the channel number is an integer, the type of the parameter would be **int**, and the declaration of the `changeChannel()` method might look like

```
public void changeChannel(int channelNum) { ... }
```

This declaration specifies that `changeChannel()` has a parameter named `channelNum` of type **int**. However, `channelNum` does not yet have any particular value. A value for `channelNum` is provided when the subroutine is called; for example: `changeChannel(17);`

The parameter list in a subroutine can be empty, or it can consist of one or more parameter declarations of the form ⟨*type*⟩ ⟨*parameter-name*⟩. If there are several declarations, they are separated by commas. Note that each declaration can name only one parameter. For example, if you want two parameters of type **double**, you have to say "`double x, double y`", rather than "`double x, y`".

Parameters are covered in more detail in the next section.

Here are a few examples of subroutine definitions, leaving out the statements that define what the subroutines do:

```
public static void playGame() {
    // "public" and "static" are modifiers; "void" is the
    // return-type; "playGame" is the subroutine-name;
    // the parameter-list is empty.
    . . .  // Statements that define what playGame does go here.
}

int getNextN(int N) {
    // There are no modifiers; "int" in the return-type;
    // "getNextN" is the subroutine-name; the parameter-list
    // includes one parameter whose name is "N" and whose
    // type is "int".
    . . .  // Statements that define what getNextN does go here.
}

static boolean lessThan(double x, double y) {
    // "static" is a modifier; "boolean" is the
    // return-type; "lessThan" is the subroutine-name;
```

```
        // the parameter-list includes two parameters whose names are
        // "x" and "y", and the type of each of these parameters
        // is "double".
        . . .  // Statements that define what lessThan does go here.
    }
```

In the second example given here, `getNextN` is a non-static method, since its definition does not include the modifier "`static`"—and so it's not an example that we should be looking at in this chapter! The other modifier shown in the examples is "`public`". This modifier indicates that the method can be called from anywhere in a program, even from outside the class where the method is defined. There is another modifier, "`private`", which indicates that the method can be called **only** from inside the same class. The modifiers `public` and `private` are called ***access specifiers***. If no access specifier is given for a method, then by default, that method can be called from anywhere in the "package" that contains the class, but not from outside that package. (Packages were introduced in Subsection 2.6.4, and you'll learn more about them later in this chapter, in Section 4.5.) There is one other access modifier, `protected`, which will only become relevant when we turn to object-oriented programming in Chapter 5.

Note, by the way, that the `main()` routine of a program follows the usual syntax rules for a subroutine. In

```
public static void main(String[] args) { ... }
```

the modifiers are `public` and `static`, the return type is `void`, the subroutine name is `main`, and the parameter list is "`String[] args`". The only question might be about "`String[]`", which has to be a type if it is to match the syntax of a parameter list. In fact, `String[]` represents a so-called "array type", so the syntax is valid. We will cover arrays in Chapter 7. (The parameter, `args`, represents information provided to the program when the `main()` routine is called by the system. In case you know the term, the information consists of any "command-line arguments" specified in the command that the user typed to run the program.)

You've already had some experience with filling in the implementation of a subroutine. In this chapter, you'll learn all about writing your own complete subroutine definitions, including the interface part.

4.2.2 Calling Subroutines

When you define a subroutine, all you are doing is telling the computer that the subroutine exists and what it does. The subroutine doesn't actually get executed until it is called. (This is true even for the `main()` routine in a class—even though **you** don't call it, it is called by the system when the system runs your program.) For example, the `playGame()` method given as an example above could be called using the following subroutine call statement:

```
playGame();
```

This statement could occur anywhere in the same class that includes the definition of `playGame()`, whether in a `main()` method or in some other subroutine. Since `playGame()` is a `public` method, it can also be called from other classes, but in that case, you have to tell the computer which class it comes from. Since `playGame()` is a `static` method, its full name includes the name of the class in which it is defined. Let's say, for example, that `playGame()` is defined in a class named `Poker`. Then to call `playGame()` from **outside** the `Poker` class, you would have to say

```
Poker.playGame();
```

The use of the class name here tells the computer which class to look in to find the method. It also lets you distinguish between `Poker.playGame()` and other potential `playGame()` methods defined in other classes, such as `Roulette.playGame()` or `Blackjack.playGame()`.

More generally, a ***subroutine call statement*** for a `static` subroutine takes the form

⟨*subroutine-name*⟩(⟨*parameters*⟩);

if the subroutine that is being called is in the same class, or

⟨*class-name*⟩.⟨*subroutine-name*⟩(⟨*parameters*⟩);

if the subroutine is defined elsewhere, in a different class. (Non-static methods belong to objects rather than classes, and they are called using object names instead of class names. More on that later.) Note that the parameter list can be empty, as in the `playGame()` example, but the parentheses must be there even if there is nothing between them. The number of parameters that you provide when you call a subroutine must match the number listed in the parameter list in the subroutine definition, and the types of the parameters in the call statement must match the types in the subroutine definition.

4.2.3 Subroutines in Programs

It's time to give an example of what a complete program looks like, when it includes other subroutines in addition to the `main()` routine. Let's write a program that plays a guessing game with the user. The computer will choose a random number between 1 and 100, and the user will try to guess it. The computer tells the user whether the guess is high or low or correct. If the user gets the number after six guesses or fewer, the user wins the game. After each game, the user has the option of continuing with another game.

Since playing one game can be thought of as a single, coherent task, it makes sense to write a subroutine that will play one guessing game with the user. The `main()` routine will use a loop to call the `playGame()` subroutine over and over, as many times as the user wants to play. We approach the problem of designing the `playGame()` subroutine the same way we write a `main()` routine: Start with an outline of the algorithm and apply stepwise refinement. Here is a short pseudocode algorithm for a guessing game routine:

```
Pick a random number
while the game is not over:
    Get the user's guess
    Tell the user whether the guess is high, low, or correct.
```

The test for whether the game is over is complicated, since the game ends if either the user makes a correct guess or the number of guesses is six. As in many cases, the easiest thing to do is to use a "`while (true)`" loop and use `break` to end the loop whenever we find a reason to do so. Also, if we are going to end the game after six guesses, we'll have to keep track of the number of guesses that the user has made. Filling out the algorithm gives:

```
Let computersNumber be a random number between 1 and 100
Let guessCount = 0
while (true):
    Get the user's guess
    Count the guess by adding 1 to guess count
    if the user's guess equals computersNumber:
        Tell the user he won
        break out of the loop
    if the number of guesses is 6:
```

```
        Tell the user he lost
        break out of the loop
    if the user's guess is less than computersNumber:
        Tell the user the guess was low
    else if the user's guess is higher than computersNumber:
        Tell the user the guess was high
```

With variable declarations added and translated into Java, this becomes the definition of the `playGame()` routine. A random integer between 1 and 100 can be computed as `(int)(100 * Math.random()) + 1`. I've cleaned up the interaction with the user to make it flow better.

```
static void playGame() {
    int computersNumber; // A random number picked by the computer.
    int usersGuess;      // A number entered by user as a guess.
    int guessCount;      // Number of guesses the user has made.
    computersNumber = (int)(100 * Math.random()) + 1;
             // The value assigned to computersNumber is a randomly
             //    chosen integer between 1 and 100, inclusive.
    guessCount = 0;
    TextIO.putln();
    TextIO.put("What is your first guess? ");
    while (true) {
       usersGuess = TextIO.getInt();  // Get the user's guess.
       guessCount++;
       if (usersGuess == computersNumber) {
          TextIO.putln("You got it in " + guessCount
                  + " guesses!  My number was " + computersNumber);
          break;  // The game is over; the user has won.
       }
       if (guessCount == 6) {
          TextIO.putln("You didn't get the number in 6 guesses.");
          TextIO.putln("You lose.  My number was " + computersNumber);
          break;  // The game is over; the user has lost.
       }
       // If we get to this point, the game continues.
       // Tell the user if the guess was too high or too low.
       if (usersGuess < computersNumber)
          TextIO.put("That's too low.  Try again: ");
       else if (usersGuess > computersNumber)
          TextIO.put("That's too high.  Try again: ");
    }
    TextIO.putln();
} // end of playGame()
```

Now, where exactly should you put this? It should be part of the same class as the `main()` routine, but **not** inside the main routine. It is not legal to have one subroutine physically nested inside another. The `main()` routine will **call** `playGame()`, but not contain it physically. You can put the definition of `playGame()` either before or after the `main()` routine. Java is not very picky about having the members of a class in any particular order.

It's pretty easy to write the main routine. You've done things like this before. Here's what the complete program looks like (except that a serious program needs more comments than I've included here).

```
public class GuessingGame {

   public static void main(String[] args) {
      TextIO.putln("Let's play a game.  I'll pick a number between");
      TextIO.putln("1 and 100, and you try to guess it.");
      boolean playAgain;
      do {
         playGame();  // call subroutine to play one game
         TextIO.put("Would you like to play again? ");
         playAgain = TextIO.getlnBoolean();
      } while (playAgain);
      TextIO.putln("Thanks for playing.  Goodbye.");
   } // end of main()

   static void playGame() {
       int computersNumber; // A random number picked by the computer.
       int usersGuess;      // A number entered by user as a guess.
       int guessCount;      // Number of guesses the user has made.
       computersNumber = (int)(100 * Math.random()) + 1;
                // The value assigned to computersNumber is a randomly
                //    chosen integer between 1 and 100, inclusive.
       guessCount = 0;
       TextIO.putln();
       TextIO.put("What is your first guess? ");
       while (true) {
          usersGuess = TextIO.getInt();  // Get the user's guess.
          guessCount++;
          if (usersGuess == computersNumber) {
             TextIO.putln("You got it in " + guessCount
                     + " guesses!  My number was " + computersNumber);
             break;  // The game is over; the user has won.
          }
          if (guessCount == 6) {
             TextIO.putln("You didn't get the number in 6 guesses.");
             TextIO.putln("You lose.  My number was " + computersNumber);
             break;  // The game is over; the user has lost.
          }
          // If we get to this point, the game continues.
          // Tell the user if the guess was too high or too low.
          if (usersGuess < computersNumber)
             TextIO.put("That's too low.  Try again: ");
          else if (usersGuess > computersNumber)
             TextIO.put("That's too high.  Try again: ");
       }
       TextIO.putln();
   } // end of playGame()

} // end of class GuessingGame
```

Take some time to read the program carefully and figure out how it works. And try to convince yourself that even in this relatively simple case, breaking up the program into two methods makes the program easier to understand and probably made it easier to write each piece.

4.2.4 Member Variables

A class can include other things besides subroutines. In particular, it can also include variable declarations. Of course, you can declare variables **inside** subroutines. Those are called ***local variables.*** However, you can also have variables that are not part of any subroutine. To distinguish such variables from local variables, we call them ***member variables***, since they are members of a class.

Just as with subroutines, member variables can be either static or non-static. In this chapter, we'll stick to static variables. A static member variable belongs to the class itself, and it exists as long as the class exists. Memory is allocated for the variable when the class is first loaded by the Java interpreter. Any assignment statement that assigns a value to the variable changes the content of that memory, no matter where that assignment statement is located in the program. Any time the variable is used in an expression, the value is fetched from that same memory, no matter where the expression is located in the program. This means that the value of a static member variable can be set in one subroutine and used in another subroutine. Static member variables are "shared" by all the static subroutines in the class. A local variable in a subroutine, on the other hand, exists only while that subroutine is being executed, and is completely inaccessible from outside that one subroutine.

The declaration of a member variable looks just like the declaration of a local variable except for two things: The member variable is declared outside any subroutine (although it still has to be inside a class), and the declaration can be marked with modifiers such as `static`, `public`, and `private`. Since we are only working with static member variables for now, every declaration of a member variable in this chapter will include the modifier `static`. They might also be marked as `public` or `private`. For example:

```
static String usersName;
public static int numberOfPlayers;
private static double velocity, time;
```

A static member variable that is not declared to be `private` can be accessed from outside the class where it is defined, as well as inside. When it is used in some other class, it must be referred to with a compound identifier of the form ⟨*class-name*⟩.⟨*variable-name*⟩. For example, the *System* class contains the public static member variable named `out`, and you use this variable in your own classes by referring to `System.out`. Similarly, `Math.PI` is a public member variable in the *Math* whose value is the mathematical constant π. If `numberOfPlayers` is a public static member variable in a class named `Poker`, then subroutines in the `Poker` class would refer to it simply as `numberOfPlayers`, while subroutines in another class would refer to it as `Poker.numberOfPlayers`.

As an example, let's add a static member variable to the `GuessingGame` class that we wrote earlier in this section. This variable will be used to keep track of how many games the user wins. We'll call the variable `gamesWon` and declare it with the statement "`static int gamesWon;`". In the `playGame()` routine, we add 1 to `gamesWon` if the user wins the game. At the end of the `main()` routine, we print out the value of `gamesWon`. It would be impossible to do the same thing with a local variable, since we need access to the same variable from both subroutines.

When you declare a local variable in a subroutine, you have to assign a value to that variable before you can do anything with it. Member variables, on the other hand are automatically initialized with a default value. For numeric variables, the default value is zero. For **boolean** variables, the default is `false`. And for **char** variables, it's the unprintable character that has Unicode code number zero. (For objects, such as `Strings`, the default initial value is a special

value called `null`, which we won't encounter officially until later.)

Since it is of type **int**, the static member variable `gamesWon` automatically gets assigned an initial value of zero. This happens to be the correct initial value for a variable that is being used as a counter. You can, of course, assign a different value to the variable at the beginning of the `main()` routine if you are not satisfied with the default initial value.

Here's a revised version of `GuessingGame.java` that includes the `gamesWon` variable. The changes from the above version are shown in italic:

```
public class GuessingGame2 {

    static int gamesWon;      // The number of games won by
                              //    the user.

    public static void main(String[] args) {
       gamesWon = 0;  // This is actually redundant, since 0 is
                      //                  the default initial value.
       TextIO.putln("Let's play a game.  I'll pick a number between");
       TextIO.putln("1 and 100, and you try to guess it.");
       boolean playAgain;
       do {
          playGame();  // call subroutine to play one game
          TextIO.put("Would you like to play again? ");
          playAgain = TextIO.getlnBoolean();
       } while (playAgain);
       TextIO.putln();
       TextIO.putln("You won " + gamesWon + " games.");
       TextIO.putln("Thanks for playing.  Goodbye.");
    } // end of main()

    static void playGame() {
        int computersNumber; // A random number picked by the computer.
        int usersGuess;      // A number entered by user as a guess.
        int guessCount;      // Number of guesses the user has made.
        computersNumber = (int)(100 * Math.random()) + 1;
                 // The value assigned to computersNumber is a randomly
                 //    chosen integer between 1 and 100, inclusive.
        guessCount = 0;
        TextIO.putln();
        TextIO.put("What is your first guess? ");
        while (true) {
           usersGuess = TextIO.getInt();  // Get the user's guess.
           guessCount++;
           if (usersGuess == computersNumber) {
              TextIO.putln("You got it in " + guessCount
                      + " guesses!  My number was " + computersNumber);
              gamesWon++;  // Count this game by incrementing gamesWon.
              break;       // The game is over; the user has won.
           }
           if (guessCount == 6) {
              TextIO.putln("You didn't get the number in 6 guesses.");
              TextIO.putln("You lose.  My number was " + computersNumber);
              break;  // The game is over; the user has lost.
           }
           // If we get to this point, the game continues.
```

```
            // Tell the user if the guess was too high or too low.
            if (usersGuess < computersNumber)
               TextIO.put("That's too low.  Try again: ");
            else if (usersGuess > computersNumber)
               TextIO.put("That's too high.  Try again: ");
         }
         TextIO.putln();
      } // end of playGame()

} // end of class GuessingGame2
```

4.3 Parameters

If a subroutine is a black box, then a parameter is something that provides a mechanism for passing information from the outside world into the box. Parameters are part of the interface of a subroutine. They allow you to customize the behavior of a subroutine to adapt it to a particular situation.

As an analogy, consider a thermostat—a black box whose task it is to keep your house at a certain temperature. The thermostat has a parameter, namely the dial that is used to set the desired temperature. The thermostat always performs the same task: maintaining a constant temperature. However, the exact task that it performs—that is, **which** temperature it maintains—is customized by the setting on its dial.

4.3.1 Using Parameters

As an example, let's go back to the "3N+1" problem that was discussed in Subsection 3.2.2. (Recall that a 3N+1 sequence is computed according to the rule, "if N is odd, multiply it by 3 and add 1; if N is even, divide it by 2; continue until N is equal to 1." For example, starting from N=3 we get the sequence: 3, 10, 5, 16, 8, 4, 2, 1.) Suppose that we want to write a subroutine to print out such sequences. The subroutine will always perform the same task: Print out a 3N+1 sequence. But the exact sequence it prints out depends on the starting value of N. So, the starting value of N would be a parameter to the subroutine. The subroutine could be written like this:

```
/**
 * This subroutine prints a 3N+1 sequence to standard output, using
 * startingValue as the initial value of N.  It also prints the number
 * of terms in the sequence. The value of the parameter, startingValue,
 * must  be a positive integer.
 */

static void print3NSequence(int startingValue) {

   int N;      // One of the terms in the sequence.
   int count;  // The number of terms.

   N = startingValue;  // The first term is whatever value
                       //    is passed to the subroutine as
                       //    a parameter.

   count = 1; // We have one term, the starting value, so far.

   System.out.println("The 3N+1 sequence starting from " + N);
```

```
        System.out.println();
        System.out.println(N);  // print initial term of sequence

        while (N > 1) {
            if (N % 2 == 1)     // is N odd?
               N = 3 * N + 1;
            else
               N = N / 2;
            count++;   // count this term
            System.out.println(N);  // print this term
        }

        System.out.println();
        System.out.println("There were " + count + " terms in the sequence.");

    }  // end print3NSequence
```

The parameter list of this subroutine, "`(int startingValue)`", specifies that the subroutine has one parameter, of type **int**. Within the body of the subroutine, the parameter name can be used in the same way as a variable name. However, the parameter gets its initial value from **outside** the subroutine. When the subroutine is called, a value must be provided for this parameter in the subroutine call statement. This value will be assigned to the parameter `startingValue` before the body of the subroutine is executed. For example, the subroutine could be called using the subroutine call statement "`print3NSequence(17);`". When the computer executes this statement, the computer first assigns the value 17 to `startingValue` and then executes the statements in the subroutine. This prints the 3N+1 sequence starting from 17. If `K` is a variable of type **int**, then when the computer executes the subroutine call statement "`print3NSequence(K);`", it will take the value of the variable `K`, assign that value to `startingValue`, and execute the body of the subroutine.

The class that contains `print3NSequence` can contain a `main()` routine (or other subroutines) that call `print3NSequence`. For example, here is a `main()` program that prints out 3N+1 sequences for various starting values specified by the user:

```
    public static void main(String[] args) {
       System.out.println("This program will print out 3N+1 sequences");
       System.out.println("for starting values that you specify.");
       System.out.println();
       int K;  // Input from user; loop ends when K < 0.
       do {
          System.out.println("Enter a starting value.");
          System.out.print("To end the program, enter 0: ");
          K = TextIO.getInt();  // Get starting value from user.
          if (K > 0)   // Print sequence, but only if K is > 0.
             print3NSequence(K);
       } while (K > 0);   // Continue only if K > 0.
    } // end main
```

Remember that before you can use this program, the definitions of `main` and of `print3NSequence` must both be wrapped inside a class definition.

4.3.2 Formal and Actual Parameters

Note that the term "parameter" is used to refer to two different, but related, concepts. There are parameters that are used in the definitions of subroutines, such as `startingValue` in the

above example. And there are parameters that are used in subroutine call statements, such as the `K` in the statement "`print3NSequence(K);`". Parameters in a subroutine definition are called ***formal parameters*** or ***dummy parameters***. The parameters that are passed to a subroutine when it is called are called ***actual parameters*** or ***arguments***. When a subroutine is called, the actual parameters in the subroutine call statement are evaluated and the values are assigned to the formal parameters in the subroutine's definition. Then the body of the subroutine is executed.

A formal parameter must be a **name**, that is, a simple identifier. A formal parameter is very much like a variable, and—like a variable—it has a specified type such as **int**, **boolean**, or *String*. An actual parameter is a **value**, and so it can be specified by any expression, provided that the expression computes a value of the correct type. The type of the actual parameter must be one that could legally be assigned to the formal parameter with an assignment statement. For example, if the formal parameter is of type **double**, then it would be legal to pass an **int** as the actual parameter since **int**s can legally be assigned to **double**s. When you call a subroutine, you must provide one actual parameter for each formal parameter in the subroutine's definition. Consider, for example, a subroutine

```
static void doTask(int N, double x, boolean test) {
    // statements to perform the task go here
}
```

This subroutine might be called with the statement

```
doTask(17, Math.sqrt(z+1), z >= 10);
```

When the computer executes this statement, it has essentially the same effect as the block of statements:

```
{
  int N;       // Allocate memory locations for the formal parameters.
  double x;
  boolean test;
  N = 17;              // Assign 17 to the first formal parameter, N.
  x = Math.sqrt(z+1);  // Compute Math.sqrt(z+1), and assign it to
                       //    the second formal parameter, x.
  test = (z >= 10);    // Evaluate "z >= 10" and assign the resulting
                       //     true/false value to the third formal
                       //     parameter, test.
   // statements to perform the task go here
}
```

(There are a few technical differences between this and "`doTask(17,Math.sqrt(z+1),z>=10);`" —besides the amount of typing—because of questions about scope of variables and what happens when several variables or parameters have the same name.)

Beginning programming students often find parameters to be surprisingly confusing. Calling a subroutine that already exists is not a problem—the idea of providing information to the subroutine in a parameter is clear enough. Writing the subroutine definition is another matter. A common beginner's mistake is to assign values to the formal parameters at the beginning of the subroutine, or to ask the user to input their values. **This represents a fundamental misunderstanding.** When the statements in the subroutine are executed, the formal parameters have already been assigned initial values! The values come from the subroutine call statement. Remember that a subroutine is not independent. It is called by some other routine, and it is the calling routine's responsibility to provide appropriate values for the parameters.

4.3.3 Overloading

In order to call a subroutine legally, you need to know its name, you need to know how many formal parameters it has, and you need to know the type of each parameter. This information is called the subroutine's ***signature***. The signature of the subroutine `doTask`, used as an example above, can be expressed as: `doTask(int,double,boolean)`. Note that the signature does **not** include the names of the parameters; in fact, if you just want to **use** the subroutine, you don't even need to know what the formal parameter names are, so the names are not part of the interface.

Java is somewhat unusual in that it allows two different subroutines in the same class to have the same name, provided that their signatures are different. (The language C++ on which Java is based also has this feature.) When this happens, we say that the name of the subroutine is ***overloaded*** because it has several different meanings. The computer doesn't get the subroutines mixed up. It can tell which one you want to call by the number and types of the actual parameters that you provide in the subroutine call statement. You have already seen overloading used with *System.out*. This object includes many different methods named `println`, for example. These methods all have different signatures, such as:

```
println(int)                 println(double)
println(String)              println(char)
println(boolean)             println()
```

The computer knows which of these subroutines you want to use based on the type of the actual parameter that you provide. `System.out.println(17)` calls the subroutine with signature `println(int)`, while `System.out.println("Hello")` calls the subroutine with signature `println(String)`. Of course all these different subroutines are semantically related, which is why it is acceptable programming style to use the same name for them all. But as far as the computer is concerned, printing out an **int** is very different from printing out a *String*, which is different from printing out a **boolean**, and so forth—so that each of these operations requires a different method.

Note, by the way, that the signature does **not** include the subroutine's return type. It is illegal to have two subroutines in the same class that have the same signature but that have different return types. For example, it would be a syntax error for a class to contain two methods defined as:

```
int    getln() { ... }
double getln() { ... }
```

So it should be no surprise that in the *TextIO* class, the methods for reading different types are not all named `getln()`. In a given class, there can only be one routine that has the name `getln` and has no parameters. So, the input routines in *TextIO* are distinguished by having different names, such as `getlnInt()` and `getlnDouble()`.

Java 5.0 introduced another complication: It is possible to have a single subroutine that takes a variable number of actual parameters. You have already used subroutines that do this—the formatted output routines `System.out.printf` and `TextIO.putf`. When you call these subroutines, the number of parameters in the subroutine call can be arbitrarily large, so it would be impossible to have different subroutines to handle each case. Unfortunately, writing the definition of such a subroutine requires some knowledge of arrays, which will not be covered until Chapter 7. When we get to that chapter, you'll learn how to write subroutines with a variable number of parameters. For now, we will ignore this complication.

4.3.4 Subroutine Examples

Let's do a few examples of writing small subroutines to perform assigned tasks. Of course, this is only one side of programming with subroutines. The task performed by a subroutine is always a subtask in a larger program. The art of designing those programs—of deciding how to break them up into subtasks—is the other side of programming with subroutines. We'll return to the question of program design in Section 4.6.

As a first example, let's write a subroutine to compute and print out all the divisors of a given positive integer. The integer will be a parameter to the subroutine. Remember that the syntax of any subroutine is:

```
⟨modifiers⟩ ⟨return-type⟩ ⟨subroutine-name⟩ ( ⟨parameter-list⟩ ) {
    ⟨statements⟩
}
```

Writing a subroutine always means filling out this format. In this case, the statement of the problem tells us that there is one parameter, of type **int**, and it tells us what the statements in the body of the subroutine should do. Since we are only working with static subroutines for now, we'll need to use `static` as a modifier. We could add an access modifier (`public` or `private`), but in the absence of any instructions, I'll leave it out. Since we are not told to return a value, the return type is `void`. Since no names are specified, we'll have to make up names for the formal parameter and for the subroutine itself. I'll use `N` for the parameter and `printDivisors` for the subroutine name. The subroutine will look like

```
static void printDivisors( int N ) {
    ⟨statements⟩
}
```

and all we have left to do is to write the statements that make up the body of the routine. This is not difficult. Just remember that you have to write the body assuming that `N` already has a value! The algorithm is: "For each possible divisor `D` in the range from `1` to `N`, if `D` evenly divides `N`, then print `D`." Written in Java, this becomes:

```
/**
 * Print all the divisors of N.
 * We assume that N is a positive integer.
 */

static void printDivisors( int N ) {
    int D;   // One of the possible divisors of N.
    System.out.println("The divisors of " + N + " are:");
    for ( D = 1; D <= N; D++ ) {
       if ( N % D == 0 )  // Dose D evenly divide N?
          System.out.println(D);
    }
}
```

I've added a comment before the subroutine definition indicating the contract of the subroutine—that is, what it does and what assumptions it makes. The contract includes the assumption that `N` is a positive integer. It is up to the caller of the subroutine to make sure that this assumption is satisfied.

As a second short example, consider the problem: Write a subroutine named `printRow`. It should have a parameter `ch` of type **char** and a parameter `N` of type **int**. The subroutine should print out a line of text containing `N` copies of the character `ch`.

Here, we are told the name of the subroutine and the names of the two parameters, so we don't have much choice about the first line of the subroutine definition. The task in this case is pretty simple, so the body of the subroutine is easy to write. The complete subroutine is given by

```
/**
 * Write one line of output containing N copies of the
 * character ch.  If N <= 0, an empty line is output.
 */

static void printRow( char ch, int N ) {
    int i;  // Loop-control variable for counting off the copies.
    for ( i = 1; i <= N; i++ ) {
        System.out.print( ch );
    }
    System.out.println();
}
```

Note that in this case, the contract makes no assumption about `N`, but it makes it clear what will happen in all cases, including the unexpected case that `N < 0`.

Finally, let's do an example that shows how one subroutine can build on another. Let's write a subroutine that takes a *String* as a parameter. For each character in the string, it should print a line of output containing 25 copies of that character. It should use the `printRow()` subroutine to produce the output.

Again, we get to choose a name for the subroutine and a name for the parameter. I'll call the subroutine `printRowsFromString` and the parameter `str`. The algorithm is pretty clear: For each position `i` in the string `str`, call `printRow(str.charAt(i),25)` to print one line of the output. So, we get:

```
/**
 * For each character in str, write a line of output
 * containing 25 copies of that character.
 */

static void printRowsFromString( String str ) {
    int i;  // Loop-control variable for counting off the chars.
    for ( i = 0; i < str.length(); i++ ) {
        printRow( str.charAt(i), 25 );
    }
}
```

We could use `printRowsFromString` in a `main()` routine such as

```
public static void main(String[] args) {
    String inputLine;  // Line of text input by user.
    TextIO.put("Enter a line of text: ");
    inputLine = TextIO.getln();
    TextIO.putln();
    printRowsFromString( inputLine );
}
```

Of course, the three routines, `main()`, `printRowsFromString()`, and `printRow()`, would have to be collected together inside the same class. The program is rather useless, but it does demonstrate the use of subroutines. You'll find the program in the file *RowsOfChars.java*, if you want to take a look.

4.3.5 Throwing Exceptions

I have been talking about the "contract" of a subroutine. The contract says what the subroutine will do, provided that the caller of the subroutine provides acceptable values for subroutine's parameters. The question arises, though, what should the subroutine do when the caller violates the contract by providing bad parameter values?

We've already seen that some subroutines respond to bad parameter values by throwing exceptions. (See Section 3.7.) For example, the contract of the built-in subroutine `Double.parseDouble` says that the parameter should be a string representation of a number of type **double**; if this is true, then the subroutine will convert the string into the equivalent numeric value. If the caller violates the contract by passing an invalid string as the actual parameter, the subroutine responds by throwing an exception of type *NumberFormatException*.

Many subroutines throw *IllegalArgumentExceptions* in response to bad parameter values. You might want to take this response in your own subroutines. This can be done with a ***throw statement***. An exception is an object, and in order to throw an exception, you must create an exception object. You won't officially learn how to do this until Chapter 5, but for now, you can use the following syntax for a `throw` statement that throws an *IllegalArgumentException*:

```
throw  new  IllegalArgumentException( ⟨error-message⟩ );
```

where ⟨*error-message*⟩ is a string that describes the error that has been detected. (The word "new" in this statement is what creates the object.) To use this statement in a subroutine, you would check whether the values of the parameters are legal. If not, you would throw the exception. For example, consider the `print3NSequence` subroutine from the beginning of this section. The parameter of `print3NSequence` is supposed to be a positive integer. We can modify the subroutine definition to make it throw an exception when this condition is violated:

```
static void print3NSequence(int startingValue) {

   if (startingValue <= 0)  // The contract is violated!
      throw new IllegalArgumentException( "Starting value must be positive." );
   .
   .  // (The rest of the subroutine is the same as before.)
   .
```

If the start value is bad, the computer executes the `throw` statement. This will immediately terminate the subroutine, without executing the rest of the body of the subroutine. Furthermore, the program as a whole will crash unless the exception is "caught" and handled elsewhere in the program by a `try..catch` statement, as discussed in Section 3.7.

4.3.6 Global and Local Variables

I'll finish this section on parameters by noting that we now have three different sorts of variables that can be used inside a subroutine: local variables declared in the subroutine, formal parameter names, and static member variables that are declared outside the subroutine but inside the same class as the subroutine.

Local variables have no connection to the outside world; they are purely part of the internal working of the subroutine. Parameters are used to "drop" values into the subroutine when it is called, but once the subroutine starts executing, parameters act much like local variables. Changes made inside a subroutine to a formal parameter have no effect on the rest of the program (at least if the type of the parameter is one of the primitive types—things are more complicated in the case of objects, as we'll see later).

Things are different when a subroutine uses a variable that is defined outside the subroutine. That variable exists independently of the subroutine, and it is accessible to other parts of the program, as well as to the subroutine. Such a variable is said to be ***global*** to the subroutine, as opposed to the local variables defined inside the subroutine. The scope of a global variable includes the entire class in which it is defined. Changes made to a global variable can have effects that extend outside the subroutine where the changes are made. You've seen how this works in the last example in the previous section, where the value of the global variable, `gamesWon`, is computed inside a subroutine and is used in the `main()` routine.

It's not always bad to use global variables in subroutines, but you should realize that the global variable then has to be considered part of the subroutine's interface. The subroutine uses the global variable to communicate with the rest of the program. This is a kind of sneaky, back-door communication that is less visible than communication done through parameters, and it risks violating the rule that the interface of a black box should be straightforward and easy to understand. So before you use a global variable in a subroutine, you should consider whether it's really necessary.

I don't advise you to take an absolute stand against using global variables inside subroutines. There is at least one good reason to do it: If you think of the class as a whole as being a kind of black box, it can be very reasonable to let the subroutines inside that box be a little sneaky about communicating with each other, if that will make the class as a whole look simpler from the outside.

4.4 Return Values

A SUBROUTINE THAT RETURNS A VALUE is called a ***function***. A given function can only return a value of a specified type, called the ***return type*** of the function. A function call generally occurs in a position where the computer is expecting to find a value, such as the right side of an assignment statement, as an actual parameter in a subroutine call, or in the middle of some larger expression. A boolean-valued function can even be used as the test condition in an `if`, `while`, `for` or `do..while` statement.

(It is also legal to use a function call as a stand-alone statement, just as if it were a regular subroutine. In this case, the computer ignores the value computed by the subroutine. Sometimes this makes sense. For example, the function `TextIO.getln()`, with a return type of *String*, reads and returns a line of input typed in by the user. Usually, the line that is returned is assigned to a variable to be used later in the program, as in the statement "`name = TextIO.getln();`". However, this function is also useful as a subroutine call statement "`TextIO.getln();`", which still reads all input up to and including the next carriage return. Since the return value is not assigned to a variable or used in an expression, it is simply discarded. So, the effect of the subroutine call is to read **and discard** some input. Sometimes, discarding unwanted input is exactly what you need to do.)

4.4.1 The return statement

You've already seen how functions such as `Math.sqrt()` and `TextIO.getInt()` can be used. What you haven't seen is how to write functions of your own. A function takes the same form as a regular subroutine, except that you have to specify the value that is to be returned by the subroutine. This is done with a ***return statement***, which has the following syntax:

`return` ⟨*expression*⟩ `;`

Such a `return` statement can only occur inside the definition of a function, and the type of the ⟨*expression*⟩ must match the return type that was specified for the function. (More exactly, it must be legal to assign the expression to a variable whose type is specified by the return type.) When the computer executes this `return` statement, it evaluates the expression, terminates execution of the function, and uses the value of the expression as the returned value of the function.

For example, consider the function definition

```
static double pythagoras(double x, double y) {
      // Computes the length of the hypotenuse of a right
      // triangle, where the sides of the triangle are x and y.
    return  Math.sqrt( x*x + y*y );
}
```

Suppose the computer executes the statement "`totalLength = 17 + pythagoras(12,5);`". When it gets to the term `pythagoras(12,5)`, it assigns the actual parameters `12` and `5` to the formal parameters `x` and `y` in the function. In the body of the function, it evaluates `Math.sqrt(12.0*12.0 + 5.0*5.0)`, which works out to `13.0`. This value is "returned" by the function, so the `13.0` essentially replaces the function call in the assignment statement, which then has the same effect as the statement "`totalLength = 17+`*`13.0`*". The return value is added to `17`, and the result, 30.0, is stored in the variable, `totalLength`.

Note that a `return` statement does not have to be the last statement in the function definition. At any point in the function where you know the value that you want to return, you can return it. Returning a value will end the function immediately, skipping any subsequent statements in the function. However, it must be the case that the function definitely does return some value, no matter what path the execution of the function takes through the code.

You can use a `return` statement inside an ordinary subroutine, one with declared return type "`void`". Since a void subroutine does not return a value, the `return` statement does not include an expression; it simply takes the form "`return;`". The effect of this statement is to terminate execution of the subroutine and return control back to the point in the program from which the subroutine was called. This can be convenient if you want to terminate execution somewhere in the middle of the subroutine, but `return` statements are optional in non-function subroutines. In a function, on the other hand, a return statement, with expression, is always required.

4.4.2 Function Examples

Here is a very simple function that could be used in a program to compute 3N+1 sequences. (The 3N+1 sequence problem is one we've looked at several times already, including in the previous section.) Given one term in a 3N+1 sequence, this function computes the next term of the sequence:

```
static int nextN(int currentN) {
   if (currentN % 2 == 1)     // test if current N is odd
      return 3*currentN + 1;  // if so, return this value
   else
      return currentN / 2;    // if not, return this instead
}
```

This function has two `return` statements. Exactly one of the two `return` statements is executed to give the value of the function. Some people prefer to use a single `return` statement at the

very end of the function when possible. This allows the reader to find the `return` statement easily. You might choose to write `nextN()` like this, for example:

```
static int nextN(int currentN) {
   int answer;  // answer will be the value returned
   if (currentN % 2 == 1)    // test if current N is odd
      answer = 3*currentN+1; // if so, this is the answer
   else
      answer = currentN / 2; // if not, this is the answer
   return answer;   // (Don't forget to return the answer!)
}
```

Here is a subroutine that uses this `nextN` function. In this case, the improvement from the version of this subroutine in Section 4.3 is not great, but if `nextN()` were a long function that performed a complex computation, then it would make a lot of sense to hide that complexity inside a function:

```
static void print3NSequence(int startingValue) {

   int N;       // One of the terms in the sequence.
   int count;   // The number of terms found.

   N = startingValue;   // Start the sequence with startingValue.
   count = 1;

   System.out.println("The 3N+1 sequence starting from " + N);
   System.out.println();
   System.out.println(N);  // print initial term of sequence

   while (N > 1) {
       N = nextN( N );   // Compute next term, using the function nextN.
       count++;          // Count this term.
       System.out.println(N);  // Print this term.
   }

   System.out.println();
   System.out.println("There were " + count + " terms in the sequence.");

}
```

* * *

Here are a few more examples of functions. The first one computes a letter grade corresponding to a given numerical grade, on a typical grading scale:

```
/**
 * Returns the letter grade corresponding to the numerical
 * grade that is passed to this function as a parameter.
 */

static char letterGrade(int numGrade) {
   if (numGrade >= 90)
      return 'A';   // 90 or above gets an A
   else if (numGrade >= 80)
      return 'B';   // 80 to 89 gets a B
   else if (numGrade >= 65)
      return 'C';   // 65 to 79 gets a C
   else if (numGrade >= 50)
```

```
      return 'D';   // 50 to 64 gets a D
   else
      return 'F';   // anything else gets an F

}  // end of function letterGrade
```

The type of the return value of `letterGrade()` is **char**. Functions can return values of any type at all. Here's a function whose return value is of type **boolean**. It demonstrates some interesting programming points, so you should read the comments:

```
/**
 * The function returns true if N is a prime number.  A prime number
 * is an integer greater than 1 that is not divisible by any positive
 * integer, except itself and 1.  If N has any divisor, D, in the range
 * 1 < D < N, then it has a divisor in the range 2 to Math.sqrt(N), namely
 * either D itself or N/D.  So we only test possible divisors from 2 to
 * Math.sqrt(N).
 */

static boolean isPrime(int N) {

   int divisor;  // A number we will test to see whether it evenly divides N.

   if (N <= 1)
      return false;  // No number <= 1 is a prime.

   int maxToTry;  // The largest divisor that we need to test.

   maxToTry = (int)Math.sqrt(N);
        // We will try to divide N by numbers between 2 and maxToTry.
        // If N is not evenly divisible by any of these numbers, then
        // N is prime.  (Note that since Math.sqrt(N) is defined to
        // return a value of type double, the value must be typecast
        // to type int before it can be assigned to maxToTry.)

    for (divisor = 2; divisor <= maxToTry; divisor++) {
        if ( N % divisor == 0 )  // Test if divisor evenly divides N.
           return false;         // If so, we know N is not prime.
                                 // No need to continue testing!
    }

    // If we get to this point, N must be prime.  Otherwise,
    // the function would already have been terminated by
    // a return statement in the previous loop.

    return true;  // Yes, N is prime.

}  // end of function isPrime
```

Finally, here is a function with return type *String*. This function has a *String* as parameter. The returned value is a reversed copy of the parameter. For example, the reverse of "Hello World" is "dlroW olleH". The algorithm for computing the reverse of a string, `str`, is to start with an empty string and then to append each character from `str`, starting from the last character of `str` and working backwards to the first:

```
static String reverse(String str) {
   String copy;  // The reversed copy.
   int i;        // One of the positions in str,
                 //       from str.length() - 1 down to 0.
```

```
      copy = "";    // Start with an empty string.
      for ( i = str.length() - 1;  i >= 0;  i-- ) {
               // Append i-th char of str to copy.
         copy = copy + str.charAt(i);
      }
      return copy;
   }
```

A ***palindrome*** is a string that reads the same backwards and forwards, such as "radar". The `reverse()` function could be used to check whether a string, `word`, is a palindrome by testing "`if (word.equals(reverse(word)))`".

By the way, a typical beginner's error in writing functions is to print out the answer, instead of returning it. **This represents a fundamental misunderstanding.** The task of a function is to compute a value and return it to the point in the program where the function was called. That's where the value is used. Maybe it will be printed out. Maybe it will be assigned to a variable. Maybe it will be used in an expression. But it's not for the function to decide.

4.4.3 3N+1 Revisited

I'll finish this section with a complete new version of the 3N+1 program. This will give me a chance to show the function `nextN()`, which was defined above, used in a complete program. I'll also take the opportunity to improve the program by getting it to print the terms of the sequence in columns, with five terms on each line. This will make the output more presentable. The idea is this: Keep track of how many terms have been printed on the current line; when that number gets up to 5, start a new line of output. To make the terms line up into neat columns, I use formatted output.

```
/**
 * A program that computes and displays several 3N+1 sequences.  Starting
 * values for the sequences are input by the user.  Terms in the sequence
 * are printed in columns, with five terms on each line of output.
 * After a sequence has been displayed, the number of terms in that
 * sequence is reported to the user.
 */

public class ThreeN2 {


   public static void main(String[] args) {

      TextIO.putln("This program will print out 3N+1 sequences");
      TextIO.putln("for starting values that you specify.");
      TextIO.putln();

      int K;   // Starting point for sequence, specified by the user.
      do {
         TextIO.putln("Enter a starting value;");
         TextIO.put("To end the program, enter 0: ");
         K = TextIO.getInt();   // get starting value from user
         if (K > 0)             // print sequence, but only if K is > 0
            print3NSequence(K);
      } while (K > 0);          // continue only if K > 0

   } // end main
```

```
   /**
    * print3NSequence prints a 3N+1 sequence to standard output, using
    * startingValue as the initial value of N.  It also prints the number
    * of terms in the sequence. The value of the parameter, startingValue,
    * must be a positive integer.
    */
   static void print3NSequence(int startingValue) {

      int N;       // One of the terms in the sequence.
      int count;   // The number of terms found.
      int onLine;  // The number of terms that have been output
                   //     so far on the current line.

      N = startingValue;   // Start the sequence with startingValue;
      count = 1;           // We have one term so far.

      TextIO.putln("The 3N+1 sequence starting from " + N);
      TextIO.putln();
      TextIO.put(N, 8);  // Print initial term, using 8 characters.
      onLine = 1;        // There's now 1 term on current output line.

      while (N > 1) {
          N = nextN(N);  // compute next term
          count++;   // count this term
          if (onLine == 5) {  // If current output line is full
             TextIO.putln();  // ...then output a carriage return
             onLine = 0;      // ...and note that there are no terms
                              //               on the new line.
          }
          TextIO.putf("%8d", N);  // Print this term in an 8-char column.
          onLine++;   // Add 1 to the number of terms on this line.
      }

      TextIO.putln();  // end current line of output
      TextIO.putln();  // and then add a blank line
      TextIO.putln("There were " + count + " terms in the sequence.");

   }  // end of Print3NSequence


   /**
    * nextN computes and returns the next term in a 3N+1 sequence,
    * given that the current term is currentN.
    */
   static int nextN(int currentN) {
       if (currentN % 2 == 1)
          return 3 * currentN + 1;
       else
          return currentN / 2;
   }  // end of nextN()

} // end of class ThreeN2
```

You should read this program carefully and try to understand how it works. (Try using 27 for the starting value!)

4.5 APIs, Packages, and Javadoc

As computers and their user interfaces have become easier to use, they have also become more complex for programmers to deal with. You can write programs for a simple console-style user interface using just a few subroutines that write output to the console and read the user's typed replies. A modern graphical user interface, with windows, buttons, scroll bars, menus, text-input boxes, and so on, might make things easier for the user, but it forces the programmer to cope with a hugely expanded array of possibilities. The programmer sees this increased complexity in the form of great numbers of subroutines that are provided for managing the user interface, as well as for other purposes.

4.5.1 Toolboxes

Someone who wanted to program for Macintosh computers—and to produce programs that look and behave the way users expect them to—had to deal with the Macintosh Toolbox, a collection of well over a thousand different subroutines. There are routines for opening and closing windows, for drawing geometric figures and text to windows, for adding buttons to windows, and for responding to mouse clicks on the window. There are other routines for creating menus and for reacting to user selections from menus. Aside from the user interface, there are routines for opening files and reading data from them, for communicating over a network, for sending output to a printer, for handling communication between programs, and in general for doing all the standard things that a computer has to do. Microsoft Windows provides its own set of subroutines for programmers to use, and they are quite a bit different from the subroutines used on the Mac. Linux has several different GUI toolboxes for the programmer to choose from.

The analogy of a "toolbox" is a good one to keep in mind. Every programming project involves a mixture of innovation and reuse of existing tools. A programmer is given a set of tools to work with, starting with the set of basic tools that are built into the language: things like variables, assignment statements, if statements, and loops. To these, the programmer can add existing toolboxes full of routines that have already been written for performing certain tasks. These tools, if they are well-designed, can be used as true black boxes: They can be called to perform their assigned tasks without worrying about the particular steps they go through to accomplish those tasks. The innovative part of programming is to take all these tools and apply them to some particular project or problem (word-processing, keeping track of bank accounts, processing image data from a space probe, Web browsing, computer games, ...). This is called ***applications programming***.

A software toolbox is a kind of black box, and it presents a certain interface to the programmer. This interface is a specification of what routines are in the toolbox, what parameters they use, and what tasks they perform. This information constitutes the ***API***, or ***Applications Programming Interface***, associated with the toolbox. The Macintosh API is a specification of all the routines available in the Macintosh Toolbox. A company that makes some hardware device—say a card for connecting a computer to a network—might publish an API for that device consisting of a list of routines that programmers can call in order to communicate with and control the device. Scientists who write a set of routines for doing some kind of complex computation—such as solving "differential equations," say—would provide an API to allow others to use those routines without understanding the details of the computations they perform.

* * *

The Java programming language is supplemented by a large, standard API. You've seen part of this API already, in the form of mathematical subroutines such as `Math.sqrt()`, the *String* data type and its associated routines, and the `System.out.print()` routines. The standard Java API includes routines for working with graphical user interfaces, for network communication, for reading and writing files, and more. It's tempting to think of these routines as being built into the Java language, but they are technically subroutines that have been written and made available for use in Java programs.

Java is platform-independent. That is, the same program can run on platforms as diverse as Mac OS, Windows, Linux, and others. The same Java API must work on all these platforms. But notice that it is the **interface** that is platform-independent; the **implementation** varies from one platform to another. A Java system on a particular computer includes implementations of all the standard API routines. A Java program includes only **calls** to those routines. When the Java interpreter executes a program and encounters a call to one of the standard routines, it will pull up and execute the implementation of that routine which is appropriate for the particular platform on which it is running. This is a very powerful idea. It means that you only need to learn one API to program for a wide variety of platforms.

4.5.2 Java's Standard Packages

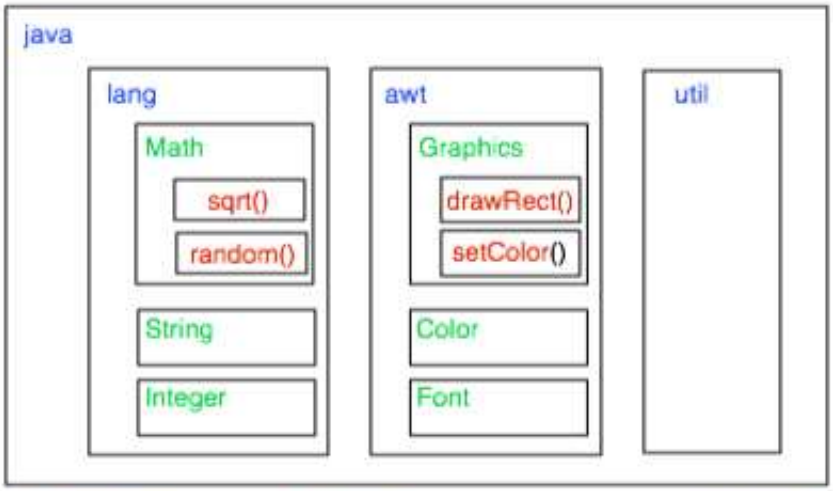
Like all subroutines in Java, the routines in the standard API are grouped into classes. To provide larger-scale organization, classes in Java can be grouped into ***packages***, which were introduced briefly in Subsection 2.6.4. You can have even higher levels of grouping, since packages can also contain other packages. In fact, the entire standard Java API is implemented in several packages. One of these, which is named "`java`", contains several non-GUI packages as well as the original AWT graphics user interface classes. Another package, "`javax`", was added in Java version 1.2 and contains the classes used by the Swing graphical user interface and other additions to the API.

A package can contain both classes and other packages. A package that is contained in another package is sometimes called a "sub-package." Both the `java` package and the `javax` package contain sub-packages. One of the sub-packages of `java`, for example, is called "`awt`". Since `awt` is contained within `java`, its full name is actually `java.awt`. This package contains classes that represent GUI components such as buttons and menus in the AWT. AWT is the older of the two Java GUI toolboxes and is no longer widely used. However, `java.awt` also contains a number of classes that form the foundation for all GUI programming, such as the `Graphics` class which provides routines for drawing on the screen, the `Color` class which represents colors, and the `Font` class which represents the fonts that are used to display characters on the screen. Since these classes are contained in the package `java.awt`, their full names are actually `java.awt.Graphics`, `java.awt.Color`, and `java.awt.Font`. (I hope that by now you've gotten the hang of how this naming thing works in Java.) Similarly, `javax` contains a sub-package named `javax.swing`, which includes such GUI classes as `javax.swing.JButton`, `javax.swing.JMenu`, and `javax.swing.JFrame`. The GUI classes in `javax.swing`, together with the foundational classes in `java.awt`, are all part of the API that makes it possible to program graphical user interfaces in Java.

The `java` package includes several other sub-packages, such as `java.io`, which provides facilities for input/output, `java.net`, which deals with network communication, and `java.util`, which provides a variety of "utility" classes. The most basic package is called `java.lang`. This package contains fundamental classes such as *String*, *Math*, *Integer*, and *Double*.

It might be helpful to look at a graphical representation of the levels of nesting in the

`java` package, its sub-packages, the classes in those sub-packages, and the subroutines in those classes. This is not a complete picture, since it shows only a very few of the many items in each element:



The official documentation for the standard Java 6 API lists 203 different packages, including sub-packages, and it lists 3793 classes in these packages. Many of these are rather obscure or very specialized, but you might want to browse through the documentation to see what is available. As I write this, the documentation for the complete API can be found at

```
http://download.oracle.com/javase/6/docs/api/
```

Even an expert programmer won't be familiar with the entire API, or even a majority of it. In this book, you'll only encounter several dozen classes, and those will be sufficient for writing a wide variety of programs.

4.5.3 Using Classes from Packages

Let's say that you want to use the class `java.awt.Color` in a program that you are writing. Like any class, `java.awt.Color` is a type, which means that you can use it to declare variables and parameters and to specify the return type of a function. One way to do this is to use the full name of the class as the name of the type. For example, suppose that you want to declare a variable named `rectColor` of type `java.awt.Color`. You could say:

```
java.awt.Color  rectColor;
```

This is just an ordinary variable declaration of the form "⟨*type-name*⟩ ⟨*variable-name*⟩;". Of course, using the full name of every class can get tiresome, so Java makes it possible to avoid using the full name of a class by ***importing*** the class. If you put

```
import java.awt.Color;
```

at the beginning of a Java source code file, then, in the rest of the file, you can abbreviate the full name `java.awt.Color` to just the simple name of the class, `Color`. Note that the `import`

line comes at the start of a file and is not inside any class. Although it is sometimes referred to as a statement, it is more properly called an ***import directive*** since it is not a statement in the usual sense. The `import` directive "`import java.awt.Color`" would allow you to say

```
Color  rectColor;
```

to declare the variable. Note that the only effect of the `import` directive is to allow you to use simple class names instead of full "package.class" names. You aren't really importing anything substantial; if you leave out the `import` directive, you can still access the class—you just have to use its full name. There is a shortcut for importing all the classes from a given package. You can import all the classes from `java.awt` by saying

```
import java.awt.*;
```

The "`*`" is a ***wildcard*** that matches every class in the package. (However, it does not match sub-packages; you **cannot** import the entire contents of all the sub-packages of the `java` package by saying `import java.*`.)

Some programmers think that using a wildcard in an `import` statement is bad style, since it can make a large number of class names available that you are not going to use and might not even know about. They think it is better to explicitly import each individual class that you want to use. In my own programming, I often use wildcards to import all the classes from the most relevant packages, and use individual imports when I am using just one or two classes from a given package.

In fact, any Java program that uses a graphical user interface is likely to use many classes from the `java.awt` and `javax.swing` packages as well as from another package named `java.awt.event`, and I often begin such programs with

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

A program that works with networking might include the line "`import java.net.*;`", while one that reads or writes files might use "`import java.io.*;`". (But when you start importing lots of packages in this way, you have to be careful about one thing: It's possible for two classes that are in different packages to have the same name. For example, both the `java.awt` package and the `java.util` package contain classes named `List`. If you import both `java.awt.*` and `java.util.*`, the simple name `List` will be ambiguous. If you try to declare a variable of type `List`, you will get a compiler error message about an ambiguous class name. The solution is simple: Use the full name of the class, either `java.awt.List` or `java.util.List`. Another solution, of course, is to use `import` to import the individual classes you need, instead of importing entire packages.)

Because the package `java.lang` is so fundamental, all the classes in `java.lang` are **automatically** imported into every program. It's as if every program began with the statement "`import java.lang.*;`". This is why we have been able to use the class name *String* instead of `java.lang.String`, and `Math.sqrt()` instead of `java.lang.Math.sqrt()`. It would still, however, be perfectly legal to use the longer forms of the names.

Programmers can create new packages. Suppose that you want some classes that you are writing to be in a package named `utilities`. Then the source code file that defines those classes must begin with the line

```
package utilities;
```

This would come even before any `import` directive in that file. Furthermore, as mentioned in Subsection 2.6.4, the source code file would be placed in a folder with the same name as the package. A class that is in a package automatically has access to other classes in the same package; that is, a class doesn't have to import the package in which it is defined.

In projects that define large numbers of classes, it makes sense to organize those classes into packages. It also makes sense for programmers to create new packages as toolboxes that provide functionality and APIs for dealing with areas not covered in the standard Java API. (And in fact such "toolmaking" programmers often have more prestige than the applications programmers who use their tools.)

However, with just a couple of exceptions, I will not be creating packages in this textbook. For the purposes of this book, you need to know about packages mainly so that you will be able to import the standard packages. These packages are always available to the programs that you write. You might wonder where the standard classes are actually located. Again, that can depend to some extent on the version of Java that you are using, but in recent standard versions, they are stored in ***jar files*** in a subdirectory named `lib` inside the Java Runtime Environment installation directory. A jar (or "Java archive") file is a single file that can contain many classes. Most of the standard classes can be found in a jar file named `rt.jar`. In fact, Java programs are generally distributed in the form of jar files, instead of as individual class files.

Although we won't be creating packages explicitly, **every** class is actually part of a package. If a class is not specifically placed in a package, then it is put in something called the ***default package***, which has no name. Almost all the examples that you see in this book are in the default package.

4.5.4 Javadoc

To use an API effectively, you need good documentation for it. The documentation for most Java APIs is prepared using a system called ***Javadoc***. For example, this system is used to prepare the documentation for Java's standard packages. And almost everyone who creates a toolbox in Java publishes Javadoc documentation for it.

Javadoc documentation is prepared from special comments that are placed in the Java source code file. Recall that one type of Java comment begins with `/*` and ends with `*/`. A Javadoc comment takes the same form, but it begins with `/**` rather than simply `/*`. You have already seen comments of this form in some of the examples in this book, such as this subroutine from Section 4.3:

```
/**
 * This subroutine prints a 3N+1 sequence to standard output, using
 * startingValue as the initial value of N.  It also prints the number
 * of terms in the sequence. The value of the parameter, startingValue,
 * must  be a positive integer.
 */

static void print3NSequence(int startingValue) { ...
```

Note that the Javadoc comment must be placed just **before** the subroutine that it is commenting on. This rule is always followed. You can have Javadoc comments for subroutines, for member variables, and for classes. The Javadoc comment always immediately **precedes** the thing it is commenting on.

Like any comment, a Javadoc comment is ignored by the computer when the file is compiled. But there is a tool called `javadoc` that reads Java source code files, extracts any Javadoc

comments that it finds, and creates a set of Web pages containing the comments in a nicely formatted, interlinked form. By default, `javadoc` will only collect information about `public` classes, subroutines, and member variables, but it allows the option of creating documentation for non-public things as well. If `javadoc` doesn't find any Javadoc comment for something, it will construct one, but the comment will contain only basic information such as the name and type of a member variable or the name, return type, and parameter list of a subroutine. This is **syntactic** information. To add information about semantics and pragmatics, you have to write a Javadoc comment.

As an example, you can look at the documentation Web page for *TextIO*. The documentation page was created by applying the `javadoc` tool to the source code file, *TextIO.java*. If you have downloaded the on-line version of this book, the documentation can be found in the `TextIO_Javadoc` directory, or you can find a link to it in the on-line version of this section.

In a Javadoc comment, the `*`'s at the start of each line are optional. The `javadoc` tool will remove them. In addition to normal text, the comment can contain certain special codes. For one thing, the comment can contain ***HTML mark-up*** commands. HTML is the language that is used to create web pages, and Javadoc comments are meant to be shown on web pages. The `javadoc` tool will copy any HTML commands in the comments to the web pages that it creates. You'll learn some basic HTML in Section 6.2, but as an example, you can add `<p>` to indicate the start of a new paragraph. (Generally, in the absence of HTML commands, blank lines and extra spaces in the comment are ignored. Furthermore, the characters & and < have special meaning in HTML and should not be used in Javadoc comments except with those meanings; they can be written as `&amp;` and `&lt;`.)

In addition to HTML commands, Javadoc comments can include ***doc tags***, which are processed as commands by the `javadoc` tool. A doc tag has a name that begins with the character `@`. I will only discuss three tags: `@param`, `@return`, and `@throws`. These tags are used in Javadoc comments for subroutines to provide information about its parameters, its return value, and the exceptions that it might throw. These tags **must** be placed at the end of the comment, after any description of the subroutine itself. The syntax for using them is:

`@param` ⟨*parameter-name*⟩ ⟨*description-of-parameter*⟩

`@return` ⟨*description-of-return-value*⟩

`@throws` ⟨*exception-class-name*⟩ ⟨*description-of-exception*⟩

The ⟨*descriptions*⟩ can extend over several lines. The description ends at the next doc tag or at the end of the comment. You can include a `@param` tag for every parameter of the subroutine and a `@throws` for as many types of exception as you want to document. You should have a `@return` tag only for a non-void subroutine. These tags do not have to be given in any particular order.

Here is an example that doesn't do anything exciting but that does use all three types of doc tag:

```
/**
 * This subroutine computes the area of a rectangle, given its width
 * and its height.  The length and the width should be positive numbers.
 * @param width the length of one side of the rectangle
 * @param height the length the second side of the rectangle
 * @return the area of the rectangle
 * @throws IllegalArgumentException if either the width or the height
 *    is a negative number.
```

```
 */
public static double areaOfRectangle( double length, double width ) {
    if ( width < 0  ||  height < 0 )
       throw new IllegalArgumentException("Sides must have positive length.");
    double area;
    area = width * height;
    return area;
}
```

I will use Javadoc comments for many of my examples. I encourage you to use them in your own code, even if you don't plan to generate Web page documentation of your work, since it's a standard format that other Java programmers will be familiar with.

If you do want to create Web-page documentation, you need to run the `javadoc` tool. This tool is available as a command in the Java Development Kit that was discussed in Section 2.6. You can use `javadoc` in a command line interface similarly to the way that the `javac` and `java` commands are used. Javadoc can also be applied in the Eclipse integrated development environment that was also discussed in Section 2.6: Just right-click the class, package, or entire project that you want to document in the Package Explorer, select "Export," and select "Javadoc" in the window that pops up. I won't go into any of the details here; see the documentation.

4.6 More on Program Design

Understanding how programs work is one thing. Designing a program to perform some particular task is another thing altogether. In Section 3.2, I discussed how pseudocode and stepwise refinement can be used to methodically develop an algorithm. We can now see how subroutines can fit into the process.

Stepwise refinement is inherently a top-down process, but the process does have a "bottom," that is, a point at which you stop refining the pseudocode algorithm and translate what you have directly into proper program code. In the absence of subroutines, the process would not bottom out until you get down to the level of assignment statements and very primitive input/output operations. But if you have subroutines lying around to perform certain useful tasks, you can stop refining as soon as you've managed to express your algorithm in terms of those tasks.

This allows you to add a bottom-up element to the top-down approach of stepwise refinement. Given a problem, you might start by writing some subroutines that perform tasks relevant to the problem domain. The subroutines become a toolbox of ready-made tools that you can integrate into your algorithm as you develop it. (Alternatively, you might be able to buy or find a software toolbox written by someone else, containing subroutines that you can use in your project as black boxes.)

Subroutines can also be helpful even in a strict top-down approach. As you refine your algorithm, you are free at any point to take any sub-task in the algorithm and make it into a subroutine. Developing that subroutine then becomes a separate problem, which you can work on separately. Your main algorithm will merely call the subroutine. This, of course, is just a way of breaking your problem down into separate, smaller problems. It is still a top-down approach because the top-down analysis of the problem tells you what subroutines to write. In the bottom-up approach, you start by writing or obtaining subroutines that are relevant to the problem domain, and you build your solution to the problem on top of that foundation of subroutines.

4.6.1 Preconditions and Postconditions

When working with subroutines as building blocks, it is important to be clear about how a subroutine interacts with the rest of the program. This interaction is specified by the ***contract*** of the subroutine, as discussed in Section 4.1. A convenient way to express the contract of a subroutine is in terms of ***preconditions*** and ***postconditions***.

A precondition of a subroutine is something that must be true when the subroutine is called, if the subroutine is to work correctly. For example, for the built-in function `Math.sqrt(x)`, a precondition is that the parameter, `x`, is greater than or equal to zero, since it is not possible to take the square root of a negative number. In terms of a contract, a precondition represents an obligation of the *caller* of the subroutine. If you call a subroutine without meeting its precondition, then there is no reason to expect it to work properly. The program might crash or give incorrect results, but you can only blame yourself, not the subroutine.

A postcondition of a subroutine represents the other side of the contract. It is something that will be true after the subroutine has run (assuming that its preconditions were met—and that there are no bugs in the subroutine). The postcondition of the function `Math.sqrt()` is that the square of the value that is returned by this function is equal to the parameter that is provided when the subroutine is called. Of course, this will only be true if the precondition—that the parameter is greater than or equal to zero—is met. A postcondition of the built-in subroutine `System.out.print(x)` is that the value of the parameter has been displayed on the screen.

Preconditions most often give restrictions on the acceptable values of parameters, as in the example of `Math.sqrt(x)`. However, they can also refer to global variables that are used in the subroutine. The postcondition of a subroutine specifies the task that it performs. For a function, the postcondition should specify the value that the function returns.

Subroutines are sometimes described by comments that explicitly specify their preconditions and postconditions. When you are given a pre-written subroutine, a statement of its preconditions and postconditions tells you how to use it and what it does. When you are assigned to write a subroutine, the preconditions and postconditions give you an exact specification of what the subroutine is expected to do. I will use this approach in the example that constitutes the rest of this section. The comments are given in the form of Javadoc comments, but I will explicitly label the preconditions and postconditions. (Many computer scientists think that new doc tags `@precondition` and `@postcondition` should be added to the Javadoc system for explicit labeling of preconditions and postconditions, but that has not yet been done.)

4.6.2 A Design Example

Let's work through an example of program design using subroutines. In this example, we will use pre-written subroutines as building blocks and we will also design new subroutines that we need to complete the project.

Suppose that I have found an already-written class called `Mosaic`. This class allows a program to work with a window that displays little colored rectangles arranged in rows and columns. The window can be opened, closed, and otherwise manipulated with static member subroutines defined in the `Mosaic` class. In fact, the class defines a toolbox or API that can be used for working with such windows. Here are some of the available routines in the API, with Javadoc-style comments:

```
/**
 * Opens a "mosaic" window on the screen.
```

```
 *
 * Precondition:   The parameters rows, cols, w, and h are positive integers.
 * Postcondition:  A window is open on the screen that can display rows and
 *                   columns of colored rectangles.  Each rectangle is w pixels
 *                   wide and h pixels high.  The number of rows is given by
 *                   the first parameter and the number of columns by the
 *                   second.  Initially, all rectangles are black.
 * Note:  The rows are numbered from 0 to rows - 1, and the columns are
 * numbered from 0 to cols - 1.
 */
public static void open(int rows, int cols, int w, int h)

/**
 * Sets the color of one of the rectangles in the window.
 *
 * Precondition:   row and col are in the valid range of row and column numbers,
 *                   and r, g, and b are in the range 0 to 255, inclusive.
 * Postcondition:  The color of the rectangle in row number row and column
 *                   number col has been set to the color specified by r, g,
 *                   and b.  r gives the amount of red in the color with 0
 *                   representing no red and 255 representing the maximum
 *                   possible amount of red.  The larger the value of r, the
 *                   more red in the color.  g and b work similarly for the
 *                   green and blue color components.
 */
public static void setColor(int row, int col, int r, int g, int b)

/**
 * Gets the red component of the color of one of the rectangles.
 *
 * Precondition:   row and col are in the valid range of row and column numbers.
 * Postcondition:  The red component of the color of the specified rectangle is
 *                   returned as an integer in the range 0 to 255 inclusive.
 */
public static int getRed(int row, int col)

/**
 * Like getRed, but returns the green component of the color.
 */
public static int getGreen(int row, int col)

/**
 * Like getRed, but returns the blue component of the color.
 */
public static int getBlue(int row, int col)

/**
 * Tests whether the mosaic window is currently open.
 *
 * Precondition:   None.
 * Postcondition:  The return value is true if the window is open when this
 *                   function is called, and it is false if the window is
 *                   closed.
```

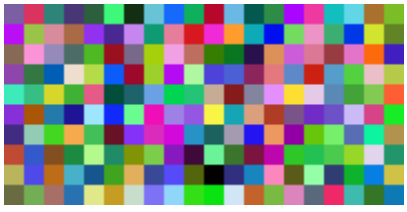
```
 */
public static boolean isOpen()

/**
 * Inserts a delay in the program (to regulate the speed at which the colors
 * are changed, for example).
 *
 * Precondition:   milliseconds is a positive integer.
 * Postcondition:  The program has paused for at least the specified number
 *                   of milliseconds, where one second is equal to 1000
 *                   milliseconds.
 */
public static void delay(int milliseconds)
```

Remember that these subroutines are members of the `Mosaic` class, so when they are called from outside `Mosaic`, the name of the class must be included as part of the name of the routine. For example, we'll have to use the name `Mosaic.isOpen()` rather than simply `isOpen()`.

You'll notice that the comments on the subroutine don't specify what happens when the preconditions are **not** met. Although a subroutine is not really obligated by its contract to do anything particular in that case, it would be good to know what happens. For example, if the precondition, "row and col are in the valid range of row and column numbers," on the `setColor()` or `getRed()` routine is violated, an *IllegalArgumentException* will be thrown. Knowing that fact would allow you to write programs that catch and handle the exception. Other questions remain about the behavior of the subroutines. For example, what happens if you call `Mosaic.open()` and there is already a mosaic window open on the screen? (In fact, the old one will be closed, and a new one will be created.) It's difficult to fully document the behavior of a piece of software—sometimes, you just have to experiment or look at the full source code.

* * *

My idea for a program is to use the `Mosaic` class as the basis for a neat animation. I want to fill the window with randomly colored squares, and then randomly change the colors in a loop that continues as long as the window is open. "Randomly change the colors" could mean a lot of different things, but after thinking for a while, I decide it would be interesting to have a "disturbance" that wanders randomly around the window, changing the color of each square that it encounters. Here's a picture showing what the contents of the window might look like at one point in time:



With basic routines for manipulating the window as a foundation, I can turn to the specific problem at hand. A basic outline for my program is

```
Open a Mosaic window
Fill window with random colors;
Move around, changing squares at random.
```

Filling the window with random colors seems like a nice coherent task that I can work on separately, so let's decide to write a separate subroutine to do it. The third step can be expanded a bit more, into the steps: Start in the middle of the window, then keep moving to new squares and changing the color of those squares. This should continue as long as the mosaic window is still open. Thus we can refine the algorithm to:

```
Open a Mosaic window
Fill window with random colors;
Set the current position to the middle square in the window;
As long as the mosaic window is open:
   Randomly change color of the square at the current position;
   Move current position up, down, left, or right, at random;
```

I need to represent the current position in some way. That can be done with two **int** variables named `currentRow` and `currentColumn` that hold the row number and the column number of the square where the disturbance is currently located. I'll use 10 rows and 20 columns of squares in my mosaic, so setting the current position to be in the center means setting `currentRow` to 5 and `currentColumn` to 10. I already have a subroutine, `Mosaic.open()`, to open the window, and I have a function, `Mosaic.isOpen()`, to test whether the window is open. To keep the main routine simple, I decide that I will write two more subroutines of my own to carry out the two tasks in the while loop. The algorithm can then be written in Java as:

```
Mosaic.open(10,20,15,15)
fillWithRandomColors();
currentRow = 5;       // Middle row, halfway down the window.
currentColumn = 10;   // Middle column.
while ( Mosaic.isOpen() ) {
    changeToRandomColor(currentRow, currentColumn);
    randomMove();
}
```

With the proper wrapper, this is essentially the `main()` routine of my program. It turns out I have to make one small modification: To prevent the animation from running too fast, the line "`Mosaic.delay(20);`" is added to the `while` loop.

The `main()` routine is taken care of, but to complete the program, I still have to write the subroutines `fillWithRandomColors()`, `changeToRandomColor(int,int)`, and `randomMove()`. Writing each of these subroutines is a separate, small task. The `fillWithRandomColors()` routine is defined by the postcondition that "each of the rectangles in the mosaic has been changed to a random color." Pseudocode for an algorithm to accomplish this task can be given as:

```
For each row:
   For each column:
      set the square in that row and column to a random color
```

"For each row" and "for each column" can be implemented as for loops. We've already planned to write a subroutine `changeToRandomColor` that can be used to set the color. (The possibility of reusing subroutines in several places is one of the big payoffs of using them!) So, `fillWithRandomColors()` can be written in proper Java as:

```
static void fillWithRandomColors() {
   for (int row = 0; row < 10; row++)
      for (int column = 0; column < 20; column++)
         changeToRandomColor(row,column);
}
```

Turning to the `changeToRandomColor` subroutine, we already have a method in the `Mosaic` class, `Mosaic.setColor()`, that can be used to change the color of a square. If we want a random color, we just have to choose random values for `r`, `g`, and `b`. According to the precondition of the `Mosaic.setColor()` subroutine, these random values must be integers in the range from 0 to 255. A formula for randomly selecting such an integer is "`(int)(256*Math.random())`". So the random color subroutine becomes:

```
static void changeToRandomColor(int rowNum, int colNum) {
     int red = (int)(256*Math.random());
     int green = (int)(256*Math.random());
     int blue = (int)(256*Math.random());
     mosaic.setColor(rowNum,colNum,red,green,blue);
}
```

Finally, consider the `randomMove` subroutine, which is supposed to randomly move the disturbance up, down, left, or right. To make a random choice among four directions, we can choose a random integer in the range 0 to 3. If the integer is 0, move in one direction; if it is 1, move in another direction; and so on. The position of the disturbance is given by the variables `currentRow` and `currentColumn`. To "move up" means to subtract 1 from `currentRow`. This leaves open the question of what to do if `currentRow` becomes -1, which would put the disturbance above the window (which would violate the precondition of several of the *Mosaic* subroutines that the row and column numbers must be in the valid range). Rather than let this happen, I decide to move the disturbance to the opposite edge of the applet by setting `currentRow` to 9. (Remember that the 10 rows are numbered from 0 to 9.) An alternative to jumping to the opposite edge would be to simply do nothing in this case. Moving the disturbance down, left, or right is handled similarly. If we use a `switch` statement to decide which direction to move, the code for `randomMove` becomes:

```
int directionNum;
directionNum = (int)(4*Math.random());
switch (directionNum) {
   case 0:  // move up
      currentRow--;
      if (currentRow < 0)   // CurrentRow is outside the mosaic;
         currentRow = 9;    // move it to the opposite edge.
      break;
   case 1:  // move right
      currentColumn++;
      if (currentColumn >= 20)
         currentColumn = 0;
      break;
   case 2:  // move down
      currentRow++;
      if (currentRow >= 10)
         currentRow = 0;
      break;
   case 3:  // move left
      currentColumn--;
      if (currentColumn < 0)
         currentColumn = 19;
      break;
}
```

4.6.3 The Program

Putting this all together, we get the following complete program. Note that I've added Javadoc-style comments for the class itself and for each of the subroutines. The variables `currentRow` and `currentColumn` are defined as static members of the class, rather than local variables, because each of them is used in several different subroutines. This program actually depends on two other classes, `Mosaic` and another class called `MosaicCanvas` that is used by `Mosaic`. If you want to compile and run this program, both of these classes must be available to the program.

```
/**
 * This program opens a window full of randomly colored squares.  A "disturbance"
 * moves randomly around in the window, randomly changing the color of each
 * square that it visits.  The program runs until the user closes the window.
 */

public class RandomMosaicWalk {

    static int currentRow;    // Row currently containing the disturbance.
    static int currentColumn; // Column currently containing disturbance.

    /**
     * The main program creates the window, fills it with random colors,
     * and then moves the disturbance in a random walk around the window
     * as long as the window is open.
     */
    public static void main(String[] args) {
        Mosaic.open(10,20,15,15);
        fillWithRandomColors();
        currentRow = 5;   // start at center of window
        currentColumn = 10;
        while (Mosaic.isOpen()) {
            changeToRandomColor(currentRow, currentColumn);
            randomMove();
            Mosaic.delay(20);
        }
    }  // end main

    /**
     * Fills the window with randomly colored squares.
     * Precondition:   The mosaic window is open.
     * Postcondition:  Each square has been set to a random color.
     */
    static void fillWithRandomColors() {
         for (int row=0; row < 10; row++) {
            for (int column=0; column < 20; column++) {
                changeToRandomColor(row, column);
            }
         }
    }  // end fillWithRandomColors

    /**
     * Changes one square to a new randomly selected color.
     * Precondition:   The specified rowNum and colNum are in the valid range
     *                 of row and column numbers.
     * Postcondition:  The square in the specified row and column has
```

```
     *                 been set to a random color.
     * @param rowNum the row number of the square, counting rows down
     *      from 0 at the top
     * @param colNum the column number of the square, counting columns over
     *      from 0 at the left
     */
    static void changeToRandomColor(int rowNum, int colNum) {
        int red, green, blue;
        red = (int)(256*Math.random());    // Choose random levels in range
        green = (int)(256*Math.random());  //     0 to 255 for red, green,
        blue = (int)(256*Math.random());   //     and blue color components.
        Mosaic.setColor(rowNum,colNum,red,green,blue);
    }  // end of changeToRandomColor()

    /**
     * Move the disturbance.
     * Precondition:   The global variables currentRow and currentColumn
     *                 are within the legal range of row and column numbers.
     * Postcondition:  currentRow or currentColumn is changed to one of the
     *                 neighboring positions in the grid -- up, down, left, or
     *                 right from the current position.  If this moves the
     *                 position outside of the grid, then it is moved to the
     *                 opposite edge of the grid.
     */
    static void randomMove() {
        int directionNum; // Randomly set to 0, 1, 2, or 3 to choose direction.
        directionNum = (int)(4*Math.random());
        switch (directionNum) {
           case 0:  // move up
              currentRow--;
              if (currentRow < 0)
                 currentRow = 9;
              break;
           case 1:  // move right
              currentColumn++;
              if (currentColumn >= 20)
                 currentColumn = 0;
              break;
           case 2:  // move down
              currentRow++;
              if (currentRow >= 10)
                 currentRow = 0;
              break;
           case 3:  // move left
              currentColumn--;
              if (currentColumn < 0)
                 currentColumn = 19;
              break;
        }
    }  // end randomMove

} // end class RandomMosaicWalk
```

4.7 The Truth About Declarations

NAMES ARE FUNDAMENTAL TO PROGRAMMING, as I said a few chapters ago. There are a lot of details involved in declaring and using names. I have been avoiding some of those details. In this section, I'll reveal most of the truth (although still not the full truth) about declaring and using variables in Java. The material in the subsections "Initialization in Declarations" and "Named Constants" is particularly important, since I will be using it regularly in future chapters.

4.7.1 Initialization in Declarations

When a variable declaration is executed, memory is allocated for the variable. This memory must be initialized to contain some definite value before the variable can be used in an expression. In the case of a local variable, the declaration is often followed closely by an assignment statement that does the initialization. For example,

```
int count;    // Declare a variable named count.
count = 0;    // Give count its initial value.
```

However, the truth about declaration statements is that it is legal to include the initialization of the variable in the declaration statement. The two statements above can therefore be abbreviated as

```
int count = 0;  // Declare count and give it an initial value.
```

The computer still executes this statement in two steps: Declare the variable `count`, then assign the value 0 to the newly created variable. The initial value does not have to be a constant. It can be any expression. It is legal to initialize several variables in one declaration statement. For example,

```
char firstInitial = 'D', secondInitial = 'E';

int x, y = 1;   // OK, but only y has been initialized!

int N = 3, M = N+2;  // OK, N is initialized
                     //        before its value is used.
```

This feature is especially common in `for` loops, since it makes it possible to declare a loop control variable at the same point in the loop where it is initialized. Since the loop control variable generally has nothing to do with the rest of the program outside the loop, it's reasonable to have its declaration in the part of the program where it's actually used. For example:

```
for ( int i = 0;  i < 10;  i++ ) {
   System.out.println(i);
}
```

Again, you should remember that this is simply an abbreviation for the following, where I've added an extra pair of braces to show that `i` is considered to be local to the `for` statement and no longer exists after the `for` loop ends:

```
{
   int i;
   for ( i = 0;  i < 10;  i++ ) {
      System.out.println(i);
   }
}
```

(You might recall, by the way, that for "for-each" loops, the special type of `for` statement that is used with enumerated types, declaring the variable in the `for` is **required**. See Subsection 3.4.4.)

A member variable can also be initialized at the point where it is declared, just as for a local variable. For example:

```
public class Bank {
   static double interestRate = 0.05;
   static int maxWithdrawal = 200;
     .
     .  // More variables and subroutines.
     .
}
```

A static member variable is created as soon as the class is loaded by the Java interpreter, and the initialization is also done at that time. In the case of member variables, this is not simply an abbreviation for a declaration followed by an assignment statement. Declaration statements are the only type of statement that can occur outside of a subroutine. Assignment statements cannot, so the following is illegal:

```
public class Bank {
   static double interestRate;
   interestRate = 0.05;  // ILLEGAL:
   .                     //    Can't be outside a subroutine!:
   .
   .
```

Because of this, declarations of member variables often include initial values. In fact, as mentioned in Subsection 4.2.4, if no initial value is provided for a member variable, then a default initial value is used. For example, when declaring an integer member variable, `count`, "`static int count;`" is equivalent to "`static int count = 0;`".

4.7.2 Named Constants

Sometimes, the value of a variable is not supposed to change after it is initialized. For example, in the above example where `interestRate` is initialized to the value `0.05`, it's quite possible that 0.05 is meant to be the value throughout the entire program. In this case, the programmer is probably defining the variable, `interestRate`, to give a meaningful name to the otherwise meaningless number, `0.05`. It's easier to understand what's going on when a program says "`principal += principal*interestRate;`" rather than "`principal += principal*0.05;`".

In Java, the modifier "`final`" can be applied to a variable declaration to ensure that the value stored in the variable cannot be changed after the variable has been initialized. For example, if the member variable `interestRate` is declared with

```
final static double interestRate = 0.05;
```

then it would be impossible for the value of `interestRate` to change anywhere else in the program. Any assignment statement that tries to assign a value to `interestRate` will be rejected by the computer as a syntax error when the program is compiled.

It is legal to apply the `final` modifier to local variables and even to formal parameters, but it is most useful for member variables. I will often refer to a static member variable that is declared to be `final` as a ***named constant***, since its value remains constant for the whole time that the program is running. The readability of a program can be greatly enhanced by

using named constants to give meaningful names to important quantities in the program. A recommended style rule for named constants is to give them names that consist entirely of upper case letters, with underscore characters to separate words if necessary. For example, the preferred style for the interest rate constant would be

```
final static double INTEREST_RATE = 0.05;
```

This is the style that is generally used in Java's standard classes, which define many named constants. For example, we have already seen that the *Math* class contains a variable `Math.PI`. This variable is declared in the *Math* class as a "public final static" variable of type **double**. Similarly, the `Color` class contains named constants such as `Color.RED` and `Color.YELLOW` which are public final static variables of type `Color`. Many named constants are created just to give meaningful names to be used as parameters in subroutine calls. For example, the standard class named `Font` contains named constants `Font.PLAIN`, `Font.BOLD`, and `Font.ITALIC`. These constants are used for specifying different styles of text when calling various subroutines in the `Font` class.

Enumerated type constants (see Subsection 2.3.3) are also examples of named constants. The enumerated type definition

```
enum Alignment { LEFT, RIGHT, CENTER }
```

defines the constants `Alignment.LEFT`, `Alignment.RIGHT`, and `Alignment.CENTER`. Technically, *Alignment* is a class, and the three constants are public final static members of that class. Defining the enumerated type is similar to defining three constants of type, say, **int**:

```
public static final int ALIGNMENT_LEFT = 0;
public static final int ALIGNMNENT_RIGHT = 1;
public static final int ALIGNMENT_CENTER = 2;
```

In fact, this is how things were generally done before the introduction of enumerated types, and it is what is done with the constants `Font.PLAIN`, `Font.BOLD`, and `Font.ITALIC` mentioned above. Using the integer constants, you could define a variable of type **int** and assign it the values `ALIGNMENT_LEFT`, `ALIGNMENT_RIGHT`, or `ALIGNMENT_CENTER` to represent different types of alignment. The only problem with this is that the computer has no way of knowing that you intend the value of the variable to represent an alignment, and it will not raise any objection if the value that is assigned to the variable is not one of the three valid alignment values.

With the enumerated type, on the other hand, the only values that can be assigned to a variable of type *Alignment* are the constant values that are listed in the definition of the enumerated type. Any attempt to assign an invalid value to the variable is a syntax error which the computer will detect when the program is compiled. This extra safety is one of the major advantages of enumerated types.

* * *

Curiously enough, one of the major reasons to use named constants is that it's easy to change the value of a named constant. Of course, the value can't change while the program is running. But between runs of the program, it's easy to change the value in the source code and recompile the program. Consider the interest rate example. It's quite possible that the value of the interest rate is used many times throughout the program. Suppose that the bank changes the interest rate and the program has to be modified. If the literal number 0.05 were used throughout the program, the programmer would have to track down each place where the interest rate is used in the program and change the rate to the new value. (This is made even harder by the fact that the number 0.05 might occur in the program with other meanings

besides the interest rate, as well as by the fact that someone might have, say, used 0.025 to represent half the interest rate.) On the other hand, if the named constant `INTEREST_RATE` is declared and used consistently throughout the program, then only the single line where the constant is initialized needs to be changed.

As an extended example, I will give a new version of the `RandomMosaicWalk` program from the previous section. This version uses named constants to represent the number of rows in the mosaic, the number of columns, and the size of each little square. The three constants are declared as `final static` member variables with the lines:

```
final static int ROWS = 30;        // Number of rows in mosaic.
final static int COLUMNS = 30;     // Number of columns in mosaic.
final static int SQUARE_SIZE = 15; // Size of each square in mosaic.
```

The rest of the program is carefully modified to use the named constants. For example, in the new version of the program, the Mosaic window is opened with the statement

```
Mosaic.open(ROWS, COLUMNS, SQUARE_SIZE, SQUARE_SIZE);
```

Sometimes, it's not easy to find all the places where a named constant needs to be used. If you don't use the named constant consistently, you've more or less defeated the purpose. It's always a good idea to run a program using several different values for any named constant, to test that it works properly in all cases.

Here is the complete new program, `RandomMosaicWalk2`, with all modifications from the previous version shown in italic. I've left out some of the comments to save space.

```
public class RandomMosaicWalk2 {

    final static int ROWS = 30;        // Number of rows in mosaic.
    final static int COLUMNS = 30;     // Number of columns in mosaic.
    final static int SQUARE_SIZE = 15; // Size of each square in mosaic.

    static int currentRow;    // Row currently containing the disturbance.
    static int currentColumn; // Column currently containing the disturbance.

    public static void main(String[] args) {
        Mosaic.open( ROWS, COLUMNS, SQUARE_SIZE, SQUARE_SIZE );
        fillWithRandomColors();
        currentRow = ROWS / 2;   // start at center of window
        currentColumn = COLUMNS / 2;
        while (Mosaic.isOpen()) {
            changeToRandomColor(currentRow, currentColumn);
            randomMove();
            Mosaic.delay(20);
        }
    }  // end main

    static void fillWithRandomColors() {
         for (int row=0; row < ROWS; row++) {
            for (int column=0; column < COLUMNS; column++) {
                changeToRandomColor(row, column);
            }
         }
    }  // end fillWithRandomColors

    static void changeToRandomColor(int rowNum, int colNum) {
         int red = (int)(256*Math.random());    // Choose random levels in range
```

```
            int green = (int)(256*Math.random());  //     0 to 255 for red, green,
            int blue = (int)(256*Math.random());   //     and blue color components.
            Mosaic.setColor(rowNum,colNum,red,green,blue);
        }  // end changeToRandomColor

        static void randomMove() {
            int directionNum; // Randomly set to 0, 1, 2, or 3 to choose direction.
            directionNum = (int)(4*Math.random());
            switch (directionNum) {
               case 0:  // move up
                  currentRow--;
                  if (currentRow < 0)
                     currentRow = ROWS - 1;
                  break;
               case 1:  // move right
                  currentColumn++;
                  if (currentColumn >= COLUMNS)
                     currentColumn = 0;
                  break;
               case 2:  // move down
                  currentRow ++;
                  if (currentRow >= ROWS)
                     currentRow = 0;
                  break;
               case 3:  // move left
                  currentColumn--;
                  if (currentColumn < 0)
                     currentColumn = COLUMNS - 1;
                  break;
            }
        }  // end randomMove

    } // end class RandomMosaicWalk2
```

4.7.3 Naming and Scope Rules

When a variable declaration is executed, memory is allocated for that variable. The variable name can be used in at least some part of the program source code to refer to that memory or to the data that is stored in the memory. The portion of the program source code where the variable name is valid is called the ***scope*** of the variable. Similarly, we can refer to the scope of subroutine names and formal parameter names.

For static member subroutines, scope is straightforward. The scope of a static subroutine is the entire source code of the class in which it is defined. That is, it is possible to call the subroutine from any point in the class, including at a point in the source code before the point where the definition of the subroutine appears. It is even possible to call a subroutine from within itself. This is an example of something called "recursion," a fairly advanced topic that we will return to in Chapter 9.

For a variable that is declared as a static member variable in a class, the situation is similar, but with one complication. It is legal to have a local variable or a formal parameter that has the same name as a member variable. In that case, within the scope of the local variable or parameter, the member variable is ***hidden***. Consider, for example, a class named `Game` that has the form:

```
public class Game {

    static int count;  // member variable

    static void playGame() {
        int count;  // local variable
          .
          .   // Some statements to define playGame()
          .
    }

    .
    .   // More variables and subroutines.
    .

}  // end Game
```

In the statements that make up the body of the `playGame()` subroutine, the name "`count`" refers to the local variable. In the rest of the `Game` class, "`count`" refers to the member variable (unless hidden by other local variables or parameters named `count`). However, there is one further complication. The member variable named `count` can also be referred to by the full name `Game.count`. Usually, the full name is only used outside the class where `count` is defined. However, there is no rule against using it inside the class. The full name, `Game.count`, can be used inside the `playGame()` subroutine to refer to the member variable instead of the local variable. So, the full scope rule is that the scope of a static member variable includes the entire class in which it is defined, but where the simple name of the member variable is hidden by a local variable or formal parameter name, the member variable must be referred to by its full name of the form ⟨*className*⟩.⟨*variableName*⟩. (Scope rules for non-static members are similar to those for static members, except that, as we shall see, non-static members cannot be used in static subroutines.)

The scope of a formal parameter of a subroutine is the block that makes up the body of the subroutine. The scope of a local variable extends from the declaration statement that defines the variable to the end of the block in which the declaration occurs. As noted above, it is possible to declare a loop control variable of a `for` loop in the `for` statement, as in "`for (int i=0; i < 10; i++)`". The scope of such a declaration is considered as a special case: It is valid only within the `for` statement and does not extend to the remainder of the block that contains the `for` statement.

It is not legal to redefine the name of a formal parameter or local variable within its scope, even in a nested block. For example, this is not allowed:

```
void  badSub(int y) {
    int x;
    while (y > 0) {
       int x;  // ERROR:  x is already defined.
          .
          .
          .
    }
 }
```

In many languages, this would be legal; the declaration of `x` in the `while` loop would hide the original declaration. It is not legal in Java; however, once the block in which a variable is declared ends, its name does become available for reuse in Java. For example:

```
void goodSub(int y) {
   while (y > 10) {
      int x;
        .
        .
        .
      // The scope of x ends here.
   }
   while (y > 0) {
      int x;  // OK: Previous declaration of x has expired.
        .
        .
        .
   }
}
```

You might wonder whether local variable names can hide subroutine names. This can't happen, for a reason that might be surprising. There is no rule that variables and subroutines have to have different names. The computer can always tell whether a name refers to a variable or to a subroutine, because a subroutine name is always followed by a left parenthesis. It's perfectly legal to have a variable called `count` and a subroutine called `count` in the same class. (This is one reason why I often write subroutine names with parentheses, as when I talk about the `main()` routine. It's a good idea to think of the parentheses as part of the name.) Even more is true: It's legal to reuse class names to name variables and subroutines. The syntax rules of Java guarantee that the computer can always tell when a name is being used as a class name. A class name is a type, and so it can be used to declare variables and formal parameters and to specify the return type of a function. This means that you could legally have a class called `Insanity` in which you declare a function

```
static  Insanity  Insanity( Insanity Insanity ) { ... }
```

The first `Insanity` is the return type of the function. The second is the function name, the third is the type of the formal parameter, and the fourth is the name of the formal parameter. However, please remember that not everything that is possible is a good idea!

Exercises for Chapter 4

1. To "capitalize" a string means to change the first letter of each word in the string to upper case (if it is not already upper case). For example, a capitalized version of "Now is the time to act!" is "Now Is The Time To Act!". Write a subroutine named `printCapitalized` that will print a capitalized version of a string to standard output. The string to be printed should be a parameter to the subroutine. Test your subroutine with a `main()` routine that gets a line of input from the user and applies the subroutine to it.

Note that a letter is the first letter of a word if it is not immediately preceded in the string by another letter. Recall that there is a standard **boolean**-valued function `Character.isLetter(char)` that can be used to test whether its parameter is a letter. There is another standard **char**-valued function, `Character.toUpperCase(char)`, that returns a capitalized version of the single character passed to it as a parameter. That is, if the parameter is a letter, it returns the upper-case version. If the parameter is not a letter, it just returns a copy of the parameter.

2. The hexadecimal digits are the ordinary, base-10 digits '0' through '9' plus the letters 'A' through 'F'. In the hexadecimal system, these digits represent the values 0 through 15, respectively. Write a function named `hexValue` that uses a `switch` statement to find the hexadecimal value of a given character. The character is a parameter to the function, and its hexadecimal value is the return value of the function. You should count lower case letters 'a' through 'f' as having the same value as the corresponding upper case letters. If the parameter is not one of the legal hexadecimal digits, return `-1` as the value of the function.

A hexadecimal integer is a sequence of hexadecimal digits, such as 34A7, FF8, 174204, or FADE. If `str` is a string containing a hexadecimal integer, then the corresponding base-10 integer can be computed as follows:

```
value = 0;
for ( i = 0; i < str.length();  i++ )
   value = value*16 + hexValue( str.charAt(i) );
```

Of course, this is not valid if `str` contains any characters that are not hexadecimal digits. Write a program that reads a string from the user. If all the characters in the string are hexadecimal digits, print out the corresponding base-10 value. If not, print out an error message.

3. Write a function that simulates rolling a pair of dice until the total on the dice comes up to be a given number. The number that you are rolling for is a parameter to the function. The number of times you have to roll the dice is the return value of the function. The parameter should be one of the possible totals: 2, 3, ..., 12. The function should throw an *IllegalArgumentException* if this is not the case. Use your function in a program that computes and prints the number of rolls it takes to get snake eyes. (Snake eyes means that the total showing on the dice is 2.)

4. This exercise builds on Exercise 4.3. Every time you roll the dice repeatedly, trying to get a given total, the number of rolls it takes can be different. The question naturally arises, what's the average number of rolls to get a given total? Write a function that performs the experiment of rolling to get a given total 10000 times. The desired total is

a parameter to the subroutine. The average number of rolls is the return value. Each individual experiment should be done by calling the function you wrote for Exercise 4.3. Now, write a main program that will call your function once for each of the possible totals (2, 3, ..., 12). It should make a table of the results, something like:

```
Total On Dice     Average Number of Rolls
-------------     -----------------------
       2                35.8382
       3                18.0607
       .                   .
       .                   .
```

5. The sample program *RandomMosaicWalk.java* from Section 4.6 shows a "disturbance" that wanders around a grid of colored squares. When the disturbance visits a square, the color of that square is changed. The applet at the bottom of Section 4.7 in the on-line version of this book shows a variation on this idea. In this applet, all the squares start out with the default color, black. Every time the disturbance visits a square, a small amount is added to the green component of the color of that square. Write a subroutine that will add 25 to the green component of one of the squares in the mosaic. The row and column numbers of the square should be given as parameters to the subroutine. Recall that you can discover the current green component of the square in row `r` and column `c` with the function call `Mosaic.getGreen(r,c)`. Use your subroutine as a substitute for the `changeToRandomColor()` subroutine in the program *RandomMosaicWalk2.java*. (This is the improved version of the program from Section 4.7 that uses named constants for the number of rows, number of columns, and square size.) Set the number of rows and the number of columns to 80. Set the square size to 5.

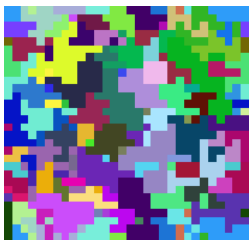
Don't forget that you will need *Mosaic.java* and *MosaicCanvas.java* to compile and run your program, since they define non-standard classes that are required by the program.

6. For this exercise, you will do something even more interesting with the *Mosaic* class that was discussed in Section 4.6. (Again, don't forget that you will need *Mosaic.java* and *MosaicCanvas.java*.)

The program that you write for this exercise should start by filling a mosaic with random colors. Then repeat the following until the user closes the mosaic window: Select one of the rectangles in the mosaic at random. Then select one of the neighboring rectangles—above it, below it, to the left of it, or to the right of it. Copy the color of the originally selected rectangle to the selected neighbor, so that the two rectangles now have the same color.

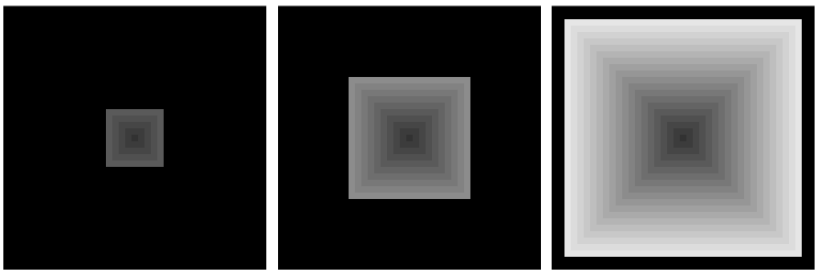
As this process is repeated over and over, it becomes more and more likely that neighboring squares will have the same color. The result is to build up larger color patches. On the other hand, once the last square of a given color disappears, there is no way for that color to ever reappear (extinction is forever!). If you let the program run long enough, eventually the entire mosaic will be one uniform color.

You can find an applet version of the program in the on-line version of this page. Here is a picture of what the mosaic looks like after the program has been running for a while:



After doing each color conversion, your program should insert a very short delay. You can try running the program without the delay; it will work, but it might be a little glitchy.

7. This is another *Mosaic* exercise, (using *Mosaic.java* and *MosaicCanvas.java* as discussed in Section 4.6). While the program does not do anything particularly interesting, it's interesting as a programming problem. An applet that does the same thing as the program can be seen in the on-line version of this book. Here is a picture showing what it looks like at several different times:



The program will show a square that grows from the center of the applet to the edges. As it grows, the part added around the edges gets brighter, so that in the end the color of the square fades from white at the edges to dark gray at the center.

The whole picture is made up of the little rectangles of a mosaic. You should first write a subroutine that draws the outline of a rectangle on a `Mosaic` window. More specifically, write a subroutine named `outlineRectangle` such that the subroutine call statement

```
outlineRectangle(top,left,height,width,r,g,b);
```

will call `Mosaic.setColor(row,col,r,g,b)` for each little square that lies on the outline of a rectangle. The topmost row of the rectangle is specified by `top`. The number of rows in the rectangle is specified by `height` (so the bottommost row is `top+height-1`). The leftmost column of the rectangle is specified by `left`. The number of columns in the rectangle is specified by `width` (so the rightmost column is `left+width-1`.) For the specific program that you are writing, the width and the height of the rectangle will always be equal, but it's nice to have the more general-purpose routine.

The animation loops through the same sequence of steps over and over. In each step, the outline of a rectangle is drawn in gray (that is, with all three color components having the same value). There is a pause of 200 milliseconds so the user can see the picture. Then the variables giving the top row, left column, size, and color level of the rectangle are adjusted to get ready for the next step. In my applet, the color level starts at 50 and increases by 10 after each step. When the rectangle gets to the outer edge of the applet, the loop ends, and the picture is erased by filling the mosaic with black. Then, after a delay of one second, the animation starts again at the beginning of the loop. You

might want to make an additional subroutine to do one loop through the steps of the basic animation.

The `main()` routine simply opens a Mosaic window and then does the animation loop over and over until the user closes the window. There is a 1000 millisecond delay between one animation loop and the next. Use a Mosaic window that has 41 rows and 41 columns. (I advise you **not** to use named constants for the numbers of rows and columns, since the problem is complicated enough already.)

Quiz on Chapter 4

1. A "black box" has an interface and an implementation. Explain what is meant by the terms *interface* and *implementation*.

2. A subroutine is said to have a *contract*. What is meant by the contract of a subroutine? When you want to use a subroutine, why is it important to understand its contract? The contract has both "syntactic" and "semantic" aspects. What is the syntactic aspect? What is the semantic aspect?

3. Briefly explain how subroutines can be useful in the top-down design of programs.

4. Discuss the concept of *parameters*. What are parameters for? What is the difference between *formal parameters* and *actual parameters*?

5. Give two different reasons for using named constants (declared with the `final` modifier).

6. What is an API? Give an example.

7. Write a subroutine named "stars" that will output a line of stars to standard output. (A star is the character "*".) The number of stars should be given as a parameter to the subroutine. Use a *for* loop. For example, the command "stars(20)" would output

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

8. Write a `main()` routine that uses the subroutine that you wrote for Question 7 to output 10 lines of stars with 1 star in the first line, 2 stars in the second line, and so on, as shown below.

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

9. Write a function named `countChars` that has a *String* and a **char** as parameters. The function should count the number of times the character occurs in the string, and it should return the result as the value of the function.

10. Write a subroutine with three parameters of type *int*. The subroutine should determine which of its parameters is smallest. The value of the smallest parameter should be returned as the value of the subroutine.

Chapter 5

Programming in the Large II: Objects and Classes

Whereas a subroutine represents a single task, an object can encapsulate both data (in the form of instance variables) and a number of different tasks or "behaviors" related to that data (in the form of instance methods). Therefore objects provide another, more sophisticated type of structure that can be used to help manage the complexity of large programs.

This chapter covers the creation and use of objects in Java. Section 5.5 covers the central ideas of object-oriented programming: inheritance and polymorphism. However, in this textbook, we will generally use these ideas in a limited form, by creating independent classes and building on existing classes rather than by designing entire hierarchies of classes from scratch. Section 5.6 and Section 5.7 cover some of the many details of object oriented programming in Java. Although these details are used occasionally later in the book, you might want to skim through them now and return to them later when they are actually needed.

5.1 Objects, Instance Methods, and Instance Variables

Object-oriented programming (OOP) represents an attempt to make programs more closely model the way people think about and deal with the world. In the older styles of programming, a programmer who is faced with some problem must identify a computing task that needs to be performed in order to solve the problem. Programming then consists of finding a sequence of instructions that will accomplish that task. But at the heart of object-oriented programming, instead of tasks we find objects—entities that have behaviors, that hold information, and that can interact with one another. Programming consists of designing a set of objects that somehow model the problem at hand. Software objects in the program can represent real or abstract entities in the problem domain. This is supposed to make the design of the program more natural and hence easier to get right and easier to understand.

To some extent, OOP is just a change in point of view. We can think of an object in standard programming terms as nothing more than a set of variables together with some subroutines for manipulating those variables. In fact, it is possible to use object-oriented techniques in any programming language. However, there is a big difference between a language that makes OOP possible and one that actively supports it. An object-oriented programming language such as Java includes a number of features that make it very different from a standard language. In order to make effective use of those features, you have to "orient" your thinking correctly.

5.1.1 Objects, Classes, and Instances

Objects are closely related to classes. We have already been working with classes for several chapters, and we have seen that a class can contain variables and subroutines. If an object is also a collection of variables and subroutines, how do they differ from classes? And why does it require a different type of thinking to understand and use them effectively? In the one section where we worked with objects rather than classes, Section 3.8, it didn't seem to make much difference: We just left the word "`static`" out of the subroutine definitions!

I have said that classes "describe" objects, or more exactly that the non-static portions of classes describe objects. But it's probably not very clear what this means. The more usual terminology is to say that objects ***belong to*** classes, but this might not be much clearer. (There is a real shortage of English words to properly distinguish all the concepts involved. An object certainly doesn't "belong" to a class in the same way that a member variable "belongs" to a class.) From the point of view of programming, it is more exact to say that classes are used to create objects. A class is a kind of factory—or blueprint—for constructing objects. The non-static parts of the class specify, or describe, what variables and subroutines the objects will contain. This is part of the explanation of how objects differ from classes: Objects are created and destroyed as the program runs, and there can be many objects with the same structure, if they are created using the same class.

Consider a simple class whose job is to group together a few static member variables. For example, the following class could be used to store information about the person who is using the program:

```
class UserData {
    static String name;
    static int age;
}
```

In a program that uses this class, there is only one copy of each of the variables `UserData.name` and `UserData.age`. There can only be one "user," since we only have memory space to store data about one user. The class, `UserData`, and the variables it contains exist as long as the program runs. (That is essentially what it means to be "static.") Now, consider a similar class that includes non-static variables:

```
class PlayerData {
   String name;
   int age;
}
```

In this case, there is no such variable as `PlayerData.name` or `PlayerData.age`, since `name` and `age` are not static members of `PlayerData`. So, there is nothing much in the class at all—except the potential to create objects. But, it's a lot of potential, since it can be used to create any number of objects! Each object will have its **own** variables called `name` and `age`. There can be many "players" because we can make new objects to represent new players on demand. A program might use this class to store information about multiple players in a game. Each player has a name and an age. When a player joins the game, a new `PlayerData` object can be created to represent that player. If a player leaves the game, the `PlayerData` object that represents that player can be destroyed. A system of objects in the program is being used to ***dynamically*** model what is happening in the game. You can't do this with static variables!

In Section 3.8, we worked with applets, which are objects. The reason they didn't seem to be any different from classes is because we were only working with one applet in each class that

we looked at. But one class can be used to make many applets. Think of an applet that scrolls a message across a Web page. There could be several such applets on the same page, all created from the same class. If the scrolling message in the applet is stored in a non-static variable, then each applet will have its own variable, and each applet can show a different message. The situation is even clearer if you think about windows on the screen, which, like applets, are objects. As a program runs, many windows might be opened and closed, but all those windows can belong to the same class. Here again, we have a dynamic situation where multiple objects are created and destroyed as a program runs.

* * *

An object that belongs to a class is said to be an ***instance*** of that class. The variables that the object contains are called ***instance variables***. The subroutines that the object contains are called ***instance methods***. (Recall that in the context of object-oriented programming, ***method*** is a synonym for "subroutine". From now on, since we are doing object-oriented programming, I will prefer the term "method.") For example, if the `PlayerData` class, as defined above, is used to create an object, then that object is an instance of the `PlayerData` class, and `name` and `age` are instance variables in the object. It is important to remember that the class of an object determines the **types** of the instance variables; however, the actual data is contained inside the individual objects, not the class. Thus, each object has its own set of data.

An applet that scrolls a message across a Web page might include a subroutine named `scroll()`. Since the applet is an object, this subroutine is an instance method of the applet. The source code for the method is in the class that is used to create the applet. Still, it's better to think of the instance method as belonging to the object, not to the class. The non-static subroutines in the class merely specify the instance methods that every object created from the class will contain. The `scroll()` methods in two different applets do the same thing in the sense that they both scroll messages across the screen. But there is a real difference between the two `scroll()` methods. The messages that they scroll can be different. You might say that the method definition in the class specifies what type of behavior the objects will have, but the specific behavior can vary from object to object, depending on the values of their instance variables.

As you can see, the static and the non-static portions of a class are very different things and serve very different purposes. Many classes contain only static members, or only non-static. However, it is possible to mix static and non-static members in a single class, and we'll see a few examples later in this chapter where it is reasonable to do so. You should distinguish between the **source code** for the class, and the **class itself**. The source code determines both the class and the objects that are created from that class. The "static" definitions in the source code specify the things that are part of the class itself, whereas the non-static definitions in the source code specify things that will become part of every instance object that is created from the class. By the way, static member variables and static member subroutines in a class are sometimes called ***class variables*** and ***class methods***, since they belong to the class itself, rather than to instances of that class.

5.1.2 Fundamentals of Objects

So far, I've been talking mostly in generalities, and I haven't given you much of an idea about what you have to put in a program if you want to work with objects. Let's look at a specific example to see how it works. Consider this extremely simplified version of a `Student` class,

which could be used to store information about students taking a course:

```
public class Student {

   public String name;  // Student's name.
   public double test1, test2, test3;   // Grades on three tests.

   public double getAverage() {  // compute average test grade
      return (test1 + test2 + test3) / 3;
   }

}  // end of class Student
```

None of the members of this class are declared to be `static`, so the class exists only for creating objects. This class definition says that any object that is an instance of the `Student` class will include instance variables named `name`, `test1`, `test2`, and `test3`, and it will include an instance method named `getAverage()`. The names and tests in different objects will generally have different values. When called for a particular student, the method `getAverage()` will compute an average using **that student's** test grades. Different students can have different averages. (Again, this is what it means to say that an instance method belongs to an individual object, not to the class.)

In Java, a class is a **type**, similar to the built-in types such as **int** and **boolean**. So, a class name can be used to specify the type of a variable in a declaration statement, the type of a formal parameter, or the return type of a function. For example, a program could define a variable named `std` of type `Student` with the statement

```
Student std;
```

However, declaring a variable does **not** create an object! This is an important point, which is related to this Very Important Fact:

In Java, no variable can ever hold an object.
A variable can only hold a reference to an object.

You should think of objects as floating around independently in the computer's memory. In fact, there is a special portion of memory called the ***heap*** where objects live. Instead of holding an object itself, a variable holds the information necessary to find the object in memory. This information is called a ***reference*** or ***pointer*** to the object. In effect, a reference to an object is the address of the memory location where the object is stored. When you use a variable of object type, the computer uses the reference in the variable to find the actual object.

In a program, objects are created using an operator called `new`, which creates an object and returns a reference to that object. For example, assuming that `std` is a variable of type `Student`, declared as above, the assignment statement

```
std = new Student();
```

would create a new object which is an instance of the class `Student`, and it would store a reference to that object in the variable `std`. The value of the variable is a reference, or pointer, to the object, not the object itself. It is not quite true, then, to say that the object is the "value of the variable `std`" (though sometimes it is hard to avoid using this terminology). It is certainly **not at all true** to say that the object is "stored in the variable `std`." The proper terminology is that "the variable `std` ***refers to*** or ***points to*** the object," and I will try to stick to that terminology as much as possible.

So, suppose that the variable `std` refers to an object belonging to the class `Student`. That object has instance variables `name`, `test1`, `test2`, and `test3`. These instance variables can be referred to as `std.name`, `std.test1`, `std.test2`, and `std.test3`. This follows the usual naming convention that when `B` is part of `A`, then the full name of `B` is `A.B`. For example, a program might include the lines

```
System.out.println("Hello, "  +  std.name  +  ".  Your test grades are:");
System.out.println(std.test1);
System.out.println(std.test2);
System.out.println(std.test3);
```

This would output the name and test grades from the object to which `std` refers. Similarly, `std` can be used to call the `getAverage()` instance method in the object by saying `std.getAverage()`. To print out the student's average, you could say:

```
System.out.println( "Your average is "  +  std.getAverage() );
```

More generally, you could use `std.name` any place where a variable of type *String* is legal. You can use it in expressions. You can assign a value to it. You can even use it to call subroutines from the *String* class. For example, `std.name.length()` is the number of characters in the student's name.

It is possible for a variable like `std`, whose type is given by a class, to refer to no object at all. We say in this case that `std` holds a ***null pointer*** or ***null reference***. The null pointer is written in Java as "`null`". You can store a null reference in the variable `std` by saying

```
std = null;
```

`null` is an actual value that is stored in the variable, not a pointer to something else. You could test whether the value of `std` is null by testing

```
if (std == null) . . .
```

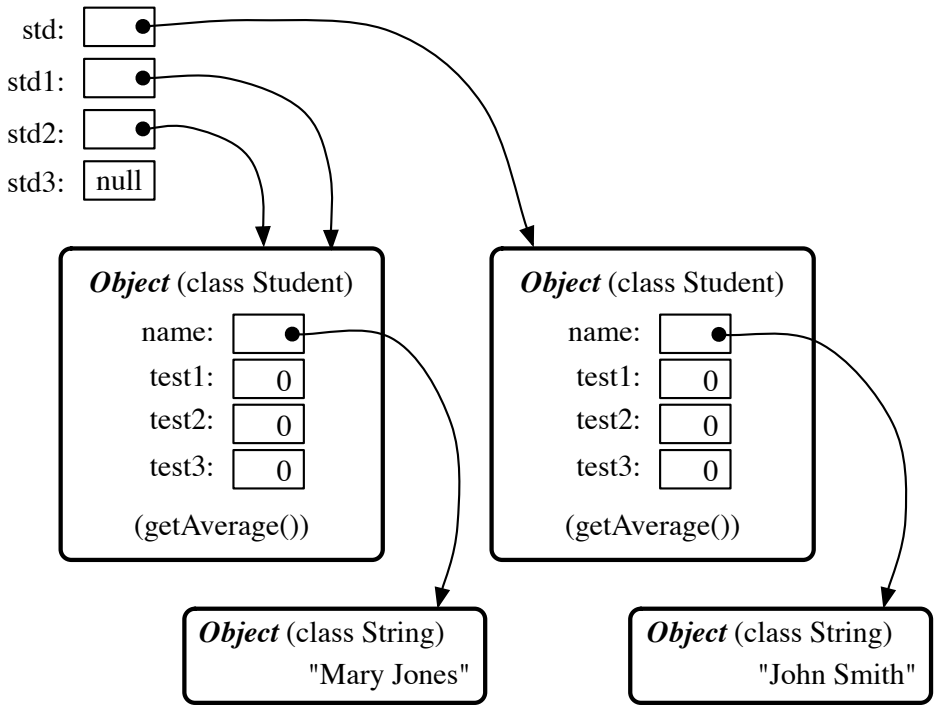
If the value of a variable is `null`, then it is, of course, illegal to refer to instance variables or instance methods through that variable—since there **is** no object, and hence no instance variables to refer to! For example, if the value of the variable `std` is `null`, then it would be illegal to refer to `std.test1`. If your program attempts to use a null pointer illegally in this way, the result is an error called a ***null pointer exception***. When this happens while the program is running, an exception of type *NullPointerException* is thrown.

Let's look at a sequence of statements that work with objects:

```
Student std, std1,       // Declare four variables of
          std2, std3;    //   type Student.
std = new Student();     // Create a new object belonging
                         //   to the class Student, and
                         //   store a reference to that
                         //   object in the variable std.
std1 = new Student();    // Create a second Student object
                         //   and store a reference to
                         //   it in the variable std1.
std2 = std1;             // Copy the reference value in std1
                         //   into the variable std2.
std3 = null;             // Store a null reference in the
                         //   variable std3.

std.name = "John Smith";  // Set values of some instance variables.
std1.name = "Mary Jones";
```

```
// (Other instance variables have default
//    initial values of zero.)
```

After the computer executes these statements, the situation in the computer's memory looks like this:



This picture shows variables as little boxes, labeled with the names of the variables. Objects are shown as boxes with round corners. When a variable contains a reference to an object, the value of that variable is shown as an arrow pointing to the object. The variable `std3`, with a value of `null`, doesn't point anywhere. The arrows from `std1` and `std2` both point to the same object. This illustrates a Very Important Point:

When one object variable is assigned to another, only a reference is copied. The object referred to is not copied.

When the assignment "`std2 = std1;`" was executed, no new object was created. Instead, `std2` was set to refer to the very same object that `std1` refers to. This is to be expected, since the assignment statement just copies the value that is stored in `std1` into `std2`, and that value is a pointer, not an object. But this has some consequences that might be surprising. For example, `std1.name` and `std2.name` are two different names for the same variable, namely the instance variable in the object that both `std1` and `std2` refer to. After the string `"Mary Jones"` is assigned to the variable `std1.name`, it is also true that the value of `std2.name` is `"Mary Jones"`. There is a potential for a lot of confusion here, but you can help protect yourself from it if you keep telling yourself, "The object is not in the variable. The variable just holds a pointer to the object."

You can test objects for equality and inequality using the operators `==` and `!=`, but here again, the semantics are different from what you are used to. When you make a test "`if (std1 == std2)`", you are testing whether the values stored in `std1` and `std2` are the same. But the values are references to objects, not objects. So, you are testing whether `std1` and `std2` refer to the same object, that is, whether they point to the same location in memory. This is fine, if its what you want to do. But sometimes, what you want to check is whether the instance variables in the objects have the same values. To do that, you would need to ask whether "`std1.test1 == std2.test1 && std1.test2 == std2.test2 && std1.test3 == std2.test3 && std1.name.equals(std2.name)`".

I've remarked previously that `Strings` are objects, and I've shown the strings `"Mary Jones"` and `"John Smith"` as objects in the above illustration. A variable of type *String* can only hold a reference to a string, not the string itself. This explains why using the `==` operator to test strings for equality is not a good idea. Suppose that `greeting` is a variable of type *String*, and that it refers to the string `"Hello"`. Then would the test `greeting == "Hello"` be true? Well, maybe, maybe not. The variable `greeting` and the *String* literal `"Hello"` each refer to a string that contains the characters H-e-l-l-o. But the strings could still be different objects, that just happen to contain the same characters, and in that case, `greeting == "Hello"` would be false. The function `greeting.equals("Hello")` tests whether `greeting` and `"Hello"` contain the same characters, which is almost certainly the question you want to ask. The expression `greeting == "Hello"` tests whether `greeting` and `"Hello"` contain the same characters **stored in the same memory location**. (Of course, a *String* variable such as `greeting` can also contain the special value `null`, and it **would** make sense to use the `==` operator to test whether "`greeting == null`".)

* * *

The fact that variables hold references to objects, not objects themselves, has a couple of other consequences that you should be aware of. They follow logically, if you just keep in mind the basic fact that the object is not stored in the variable. The object is somewhere else; the variable points to it.

Suppose that a variable that refers to an object is declared to be `final`. This means that the value stored in the variable can never be changed, once the variable has been initialized. The value stored in the variable is a reference to the object. So the variable will continue to refer to the same object as long as the variable exists. However, this does not prevent the data **in the object** from changing. The variable is `final`, not the object. It's perfectly legal to say

```
final Student stu = new Student();

stu.name = "John Doe";  // Change data in the object;
                        // The value stored in stu is not changed!
                        // It still refers to the same object.
```

Next, suppose that `obj` is a variable that refers to an object. Let's consider what happens when `obj` is passed as an actual parameter to a subroutine. The value of `obj` is assigned to a formal parameter in the subroutine, and the subroutine is executed. The subroutine has no power to change the value stored in the variable, `obj`. It only has a copy of that value. However, that value is a reference to an object. Since the subroutine has a reference to the object, it can change the data stored **in** the object. After the subroutine ends, `obj` still points to the same object, but the data stored **in the object** might have changed. Suppose `x` is a variable of type **int** and `stu` is a variable of type `Student`. Compare:

```
void dontChange(int z) {                void change(Student s) {
```

```
    z = 42;
}
```

The lines:

```
   x = 17;
   dontChange(x);
   System.out.println(x);
```

output the value 17.

The value of x is not changed by the subroutine, which is equivalent to

```
   z = x;
   z = 42;
```

```
     s.name = "Fred";
}
```

The lines:

```
   stu.name = "Jane";
   change(stu);
   System.out.println(stu.name);
```

output the value "Fred".

The value of stu is not changed, but stu.name is. This is equivalent to

```
   s = stu;
   s.name = "Fred";
```

5.1.3 Getters and Setters

When writing new classes, it's a good idea to pay attention to the issue of access control. Recall that making a member of a class `public` makes it accessible from anywhere, including from other classes. On the other hand, a `private` member can only be used in the class where it is defined.

In the opinion of many programmers, almost all member variables should be declared `private`. This gives you complete control over what can be done with the variable. Even if the variable itself is private, you can allow other classes to find out what its value is by providing a `public` ***accessor method*** that returns the value of the variable. For example, if your class contains a `private` member variable, `title`, of type *String*, you can provide a method

```
public String getTitle() {
    return title;
}
```

that returns the value of `title`. By convention, the name of an accessor method for a variable is obtained by capitalizing the name of variable and adding "get" in front of the name. So, for the variable `title`, we get an accessor method named "get" + "Title", or `getTitle()`. Because of this naming convention, accessor methods are more often referred to as ***getter methods***. A getter method provides "read access" to a variable.

You might also want to allow "write access" to a `private` variable. That is, you might want to make it possible for other classes to specify a new value for the variable. This is done with a ***setter method***. (If you don't like simple, Anglo-Saxon words, you can use the fancier term ***mutator method***.) The name of a setter method should consist of "set" followed by a capitalized copy of the variable's name, and it should have a parameter with the same type as the variable. A setter method for the variable `title` could be written

```
public void setTitle( String newTitle ) {
   title = newTitle;
}
```

It is actually very common to provide both a getter and a setter method for a private member variable. Since this allows other classes both to see and to change the value of the variable, you might wonder why not just make the variable `public`? The reason is that getters and setters are not restricted to simply reading and writing the variable's value. In fact, they

can take any action at all. For example, a getter method might keep track of the number of times that the variable has been accessed:

```
public String getTitle() {
    titleAccessCount++;  // Increment member variable titleAccessCount.
    return title;
}
```

and a setter method might check that the value that is being assigned to the variable is legal:

```
public void setTitle( String newTitle ) {
   if ( newTitle == null )   // Don't allow null strings as titles!
      title = "(Untitled)";  // Use an appropriate default value instead.
   else
      title = newTitle;
}
```

Even if you can't think of any extra chores to do in a getter or setter method, you might change your mind in the future when you redesign and improve your class. If you've used a getter and setter from the beginning, you can make the modification to your class without affecting any of the classes that use your class. The `private` member variable is not part of the public interface of your class; only the `public` getter and setter methods are, and you are free to change their implementations without changing the public interface of your class. If you **haven't** used get and set from the beginning, you'll have to contact everyone who uses your class and tell them, "Sorry guys, you'll have to track down every use that you've made of this variable and change your code to use my new get and set methods instead."

A couple of final notes: Some advanced aspects of Java rely on the naming convention for getter and setter methods, so it's a good idea to follow the convention rigorously. And though I've been talking about using getter and setter methods for a variable, you can define get and set methods even if there is no variable. A getter and/or setter method defines a ***property*** of the class, that might or might not correspond to a variable. For example, if a class includes a `public void` instance method with signature `setValue(double)`, then the class has a "property" named `value` of type **double**, and it has this property whether or not the class has a member variable named `value`.

5.2 Constructors and Object Initialization

Object types in Java are very different from the primitive types. Simply declaring a variable whose type is given as a class does not automatically create an object of that class. Objects must be explicitly ***constructed***. For the computer, the process of constructing an object means, first, finding some unused memory in the heap that can be used to hold the object and, second, filling in the object's instance variables. As a programmer, you don't care where in memory the object is stored, but you will usually want to exercise some control over what initial values are stored in a new object's instance variables. In many cases, you will also want to do more complicated initialization or bookkeeping every time an object is created.

5.2.1 Initializing Instance Variables

An instance variable can be assigned an initial value in its declaration, just like any other variable. For example, consider a class named *PairOfDice*. An object of this class will represent

a pair of dice. It will contain two instance variables to represent the numbers showing on the dice and an instance method for rolling the dice:

```
public class PairOfDice {

    public int die1 = 3;   // Number showing on the first die.
    public int die2 = 4;   // Number showing on the second die.

    public void roll() {
            // Roll the dice by setting each of the dice to be
            // a random number between 1 and 6.
         die1 = (int)(Math.random()*6) + 1;
         die2 = (int)(Math.random()*6) + 1;
    }

} // end class PairOfDice
```

The instance variables `die1` and `die2` are initialized to the values 3 and 4 respectively. These initializations are executed whenever a *PairOfDice* object is constructed. It's important to understand when and how this happens. There can be many *PairOfDice* objects. Each time one is created, it gets its own instance variables, and the assignments "`die1 = 3`" and "`die2 = 4`" are executed to fill in the values of those variables. To make this clearer, consider a variation of the *PairOfDice* class:

```
public class PairOfDice {

    public int die1 = (int)(Math.random()*6) + 1;
    public int die2 = (int)(Math.random()*6) + 1;

    public void roll() {
         die1 = (int)(Math.random()*6) + 1;
         die2 = (int)(Math.random()*6) + 1;
    }

} // end class PairOfDice
```

Here, the dice are initialized to random values, as if a new pair of dice were being thrown onto the gaming table. Since the initialization is executed for each new object, a set of random initial values will be computed for each new pair of dice. Different pairs of dice can have different initial values. For initialization of **static** member variables, of course, the situation is quite different. There is only one copy of a `static` variable, and initialization of that variable is executed just once, when the class is first loaded.

If you don't provide any initial value for an instance variable, a default initial value is provided automatically. Instance variables of numerical type (**int**, **double**, etc.) are automatically initialized to zero if you provide no other values; **boolean** variables are initialized to `false`; and **char** variables, to the Unicode character with code number zero. An instance variable can also be a variable of object type. For such variables, the default initial value is `null`. (In particular, since `Strings` are objects, the default initial value for *String* variables is `null`.)

5.2.2 Constructors

Objects are created with the operator, `new`. For example, a program that wants to use a *PairOfDice* object could say:

```
PairOfDice dice;   // Declare a variable of type PairOfDice.

dice = new PairOfDice();  // Construct a new object and store a
                          //   reference to it in the variable.
```

In this example, "`new PairOfDice()`" is an expression that allocates memory for the object, initializes the object's instance variables, and then returns a reference to the object. This reference is the value of the expression, and that value is stored by the assignment statement in the variable, `dice`, so that after the assignment statement is executed, `dice` refers to the newly created object. Part of this expression, "`PairOfDice()`", looks like a subroutine call, and that is no accident. It is, in fact, a call to a special type of subroutine called a ***constructor***. This might puzzle you, since there is no such subroutine in the class definition. However, every class has at least one constructor. If the programmer doesn't write a constructor definition in a class, then the system will provide a ***default constructor*** for that class. This default constructor does nothing beyond the basics: allocate memory and initialize instance variables. If you want more than that to happen when an object is created, you can include one or more constructors in the class definition.

The definition of a constructor looks much like the definition of any other subroutine, with three exceptions. A constructor does not have any return type (not even `void`). The name of the constructor must be the same as the name of the class in which it is defined. And the only modifiers that can be used on a constructor definition are the access modifiers `public`, `private`, and `protected`. (In particular, a constructor can't be declared `static`.)

However, a constructor does have a subroutine body of the usual form, a block of statements. There are no restrictions on what statements can be used. And it can have a list of formal parameters. In fact, the ability to include parameters is one of the main reasons for using constructors. The parameters can provide data to be used in the construction of the object. For example, a constructor for the *PairOfDice* class could provide the values that are initially showing on the dice. Here is what the class would look like in that case:

```
public class PairOfDice {

    public int die1;   // Number showing on the first die.
    public int die2;   // Number showing on the second die.

    public PairOfDice(int val1, int val2) {
            // Constructor.  Creates a pair of dice that
            // are initially showing the values val1 and val2.
         die1 = val1;  // Assign specified values
         die2 = val2;  //           to the instance variables.
    }

    public void roll() {
            // Roll the dice by setting each of the dice to be
            // a random number between 1 and 6.
         die1 = (int)(Math.random()*6) + 1;
         die2 = (int)(Math.random()*6) + 1;
    }

} // end class PairOfDice
```

The constructor is declared as "`public PairOfDice(int val1, int val2) ...`", with no return type and with the same name as the name of the class. This is how the Java compiler recognizes a constructor. The constructor has two parameters, and values for these parameters must be provided when the constructor is called. For example, the expression

"`new PairOfDice(3,4)`" would create a *PairOfDice* object in which the values of the instance variables `die1` and `die2` are initially 3 and 4. Of course, in a program, the value returned by the constructor should be used in some way, as in

```
PairOfDice dice;            // Declare a variable of type PairOfDice.

dice = new PairOfDice(1,1); // Let dice refer to a new PairOfDice
                            //   object that initially shows 1, 1.
```

Now that we've added a constructor to the *PairOfDice* class, we can no longer create an object by saying "`new PairOfDice()`"! The system provides a default constructor for a class **only** if the class definition does not already include a constructor, so there is only one constructor in the class, and it requires two actual parameters. However, this is not a big problem, since we can add a second constructor to the class, one that has no parameters. In fact, you can have as many different constructors as you want, as long as their signatures are different, that is, as long as they have different numbers or types of formal parameters. In the *PairOfDice* class, we might have a constructor with no parameters which produces a pair of dice showing random numbers:

```
public class PairOfDice {

    public int die1;   // Number showing on the first die.
    public int die2;   // Number showing on the second die.

    public PairOfDice() {
            // Constructor.  Rolls the dice, so that they initially
            // show some random values.
        roll();  // Call the roll() method to roll the dice.
    }

    public PairOfDice(int val1, int val2) {
            // Constructor.  Creates a pair of dice that
            // are initially showing the values val1 and val2.
        die1 = val1;  // Assign specified values
        die2 = val2;  //            to the instance variables.
    }

    public void roll() {
            // Roll the dice by setting each of the dice to be
            // a random number between 1 and 6.
        die1 = (int)(Math.random()*6) + 1;
        die2 = (int)(Math.random()*6) + 1;
    }

} // end class PairOfDice
```

Now we have the option of constructing a *PairOfDice* object either with "`new PairOfDice()`" or with "`new PairOfDice(x,y)`", where `x` and `y` are **int**-valued expressions.

This class, once it is written, can be used in any program that needs to work with one or more pairs of dice. None of those programs will ever have to use the obscure incantation "`(int)(Math.random()*6)+1`", because it's done inside the *PairOfDice* class. And the programmer, having once gotten the dice-rolling thing straight will never have to worry about it again. Here, for example, is a main program that uses the *PairOfDice* class to count how many times two pairs of dice are rolled before the two pairs come up showing the same value. This illustrates once again that you can create several instances of the same class:

```
public class RollTwoPairs {

    public static void main(String[] args) {

        PairOfDice firstDice;  // Refers to the first pair of dice.
        firstDice = new PairOfDice();

        PairOfDice secondDice; // Refers to the second pair of dice.
        secondDice = new PairOfDice();

        int countRolls;  // Counts how many times the two pairs of
                         //    dice have been rolled.

        int total1;      // Total showing on first pair of dice.
        int total2;      // Total showing on second pair of dice.

        countRolls = 0;

        do {  // Roll the two pairs of dice until totals are the same.

            firstDice.roll();    // Roll the first pair of dice.
            total1 = firstDice.die1 + firstDice.die2;   // Get total.
            System.out.println("First pair comes up  " + total1);

            secondDice.roll();    // Roll the second pair of dice.
            total2 = secondDice.die1 + secondDice.die2;   // Get total.
            System.out.println("Second pair comes up " + total2);

            countRolls++;   // Count this roll.

            System.out.println();  // Blank line.

        } while (total1 != total2);

        System.out.println("It took " + countRolls
                           + " rolls until the totals were the same.");

    } // end main()

} // end class RollTwoPairs
```

* * *

Constructors are subroutines, but they are subroutines of a special type. They are certainly not instance methods, since they don't belong to objects. Since they are responsible for creating objects, they exist before any objects have been created. They are more like `static` member subroutines, but they are not and cannot be declared to be `static`. In fact, according to the Java language specification, they are technically not members of the class at all! In particular, constructors are **not** referred to as "methods."

Unlike other subroutines, a constructor can only be called using the `new` operator, in an expression that has the form

`new` ⟨*class-name*⟩ `(` ⟨*parameter-list*⟩ `)`

where the ⟨*parameter-list*⟩ is possibly empty. I call this an expression because it computes and returns a value, namely a reference to the object that is constructed. Most often, you will store the returned reference in a variable, but it is also legal to use a constructor call in other ways, for example as a parameter in a subroutine call or as part of a more complex expression. Of course, if you don't save the reference in a variable, you won't have any way of referring to the object that was just created.

A constructor call is more complicated than an ordinary subroutine or function call. It is helpful to understand the exact steps that the computer goes through to execute a constructor call:

1. First, the computer gets a block of unused memory in the heap, large enough to hold an object of the specified type.
2. It initializes the instance variables of the object. If the declaration of an instance variable specifies an initial value, then that value is computed and stored in the instance variable. Otherwise, the default initial value is used.
3. The actual parameters in the constructor, if any, are evaluated, and the values are assigned to the formal parameters of the constructor.
4. The statements in the body of the constructor, if any, are executed.
5. A reference to the object is returned as the value of the constructor call.

The end result of this is that you have a reference to a newly constructed object. You can use this reference to get at the instance variables in that object or to call its instance methods.

* * *

For another example, let's rewrite the `Student` class that was used in Section 1. I'll add a constructor, and I'll also take the opportunity to make the instance variable, `name`, private.

```
public class Student {

   private String name;                 // Student's name.
   public double test1, test2, test3;   // Grades on three tests.

   Student(String theName) {
        // Constructor for Student objects;
        // provides a name for the Student.
      name = theName;
   }

   public String getName() {
        // Getter method for reading the value of the private
        // instance variable, name.
      return name;
   }

   public double getAverage() {
        // Compute average test grade.
      return (test1 + test2 + test3) / 3;
   }

}  // end of class Student
```

An object of type `Student` contains information about some particular student. The constructor in this class has a parameter of type *String*, which specifies the name of that student. Objects of type `Student` can be created with statements such as:

```
std = new Student("John Smith");
std1 = new Student("Mary Jones");
```

In the original version of this class, the value of `name` had to be assigned by a program after it created the object of type `Student`. There was no guarantee that the programmer would always remember to set the `name` properly. In the new version of the class, there is no way to

create a `Student` object except by calling the constructor, and that constructor automatically sets the `name`. The programmer's life is made easier, and whole hordes of frustrating bugs are squashed before they even have a chance to be born.

Another type of guarantee is provided by the `private` modifier. Since the instance variable, `name`, is `private`, there is no way for any part of the program outside the `Student` class to get at the `name` directly. The program sets the value of `name`, indirectly, when it calls the constructor. I've provided a getter function, `getName()`, that can be used from outside the class to find out the `name` of the student. But I haven't provided any setter method or other way to change the name. Once a student object is created, it keeps the same name as long as it exists. (It would be legal to declare the variable `name` to be "`final`" in this class. An instance variable can be `final` provided it is either assigned a value in its declaration or is assigned a value in every constructor in the class. It is illegal to assign a value to a `final` instance variable, except inside a constructor.)

5.2.3 Garbage Collection

So far, this section has been about creating objects. What about destroying them? In Java, the destruction of objects takes place automatically.

An object exists in the heap, and it can be accessed only through variables that hold references to the object. What should be done with an object if there are no variables that refer to it? Such things can happen. Consider the following two statements (though in reality, you'd never do anything like this in consecutive statements):

```
Student std = new Student("John Smith");
std = null;
```

In the first line, a reference to a newly created `Student` object is stored in the variable `std`. But in the next line, the value of `std` is changed, and the reference to the `Student` object is gone. In fact, there are now no references whatsoever to that object, in any variable. So there is no way for the program ever to use the object again! It might as well not exist. In fact, the memory occupied by the object should be reclaimed to be used for another purpose.

Java uses a procedure called ***garbage collection*** to reclaim memory occupied by objects that are no longer accessible to a program. It is the responsibility of the system, not the programmer, to keep track of which objects are "garbage." In the above example, it was very easy to see that the `Student` object had become garbage. Usually, it's much harder. If an object has been used for a while, there might be several references to the object stored in several variables. The object doesn't become garbage until all those references have been dropped.

In many other programming languages, it's the programmer's responsibility to delete the garbage. Unfortunately, keeping track of memory usage is very error-prone, and many serious program bugs are caused by such errors. A programmer might accidently delete an object even though there are still references to that object. This is called a ***dangling pointer error***, and it leads to problems when the program tries to access an object that is no longer there. Another type of error is a ***memory leak***, where a programmer neglects to delete objects that are no longer in use. This can lead to filling memory with objects that are completely inaccessible, and the program might run out of memory even though, in fact, large amounts of memory are being wasted.

Because Java uses garbage collection, such errors are simply impossible. Garbage collection is an old idea and has been used in some programming languages since the 1960s. You might wonder why all languages don't use garbage collection. In the past, it was considered too slow

and wasteful. However, research into garbage collection techniques combined with the incredible speed of modern computers have combined to make garbage collection feasible. Programmers should rejoice.

5.3 Programming with Objects

THERE ARE SEVERAL WAYS in which object-oriented concepts can be applied to the process of designing and writing programs. The broadest of these is ***object-oriented analysis and design*** which applies an object-oriented methodology to the earliest stages of program development, during which the overall design of a program is created. Here, the idea is to identify things in the problem domain that can be modeled as objects. On another level, object-oriented programming encourages programmers to produce ***generalized software components*** that can be used in a wide variety of programming projects.

Of course, for the most part, you will experience "generalized software components" by using the standard classes that come along with Java. We begin this section by looking at some built-in classes that are used for creating objects. At the end of the section, we will get back to generalities.

5.3.1 Some Built-in Classes

Although the focus of object-oriented programming is generally on the design and implementation of new classes, it's important not to forget that the designers of Java have already provided a large number of reusable classes. Some of these classes are meant to be extended to produce new classes, while others can be used directly to create useful objects. A true mastery of Java requires familiarity with a large number of built-in classes—something that takes a lot of time and experience to develop. In the next chapter, we will begin the study of Java's GUI classes, and you will encounter other built-in classes throughout the remainder of this book. But let's take a moment to look at a few built-in classes that you might find useful.

A string can be built up from smaller pieces using the `+` operator, but this is not always efficient. If `str` is a *String* and `ch` is a character, then executing the command "`str = str + ch;`" involves creating a whole new string that is a copy of `str`, with the value of `ch` appended onto the end. Copying the string takes some time. Building up a long string letter by letter would require a surprising amount of processing. The class *StringBuffer* makes it possible to be efficient about building up a long string from a number of smaller pieces. To do this, you must make an object belonging to the *StringBuffer* class. For example:

```
StringBuffer buffer = new StringBuffer();
```

(This statement both declares the variable `buffer` and initializes it to refer to a newly created *StringBuffer* object. Combining declaration with initialization was covered in Subsection 4.7.1 and works for objects just as it does for primitive types.)

Like a *String*, a *StringBuffer* contains a sequence of characters. However, it is possible to add new characters onto the end of a `StringBuffer` without making a copy of the data that it already contains. If `x` is a value of any type and `buffer` is the variable defined above, then the command `buffer.append(x)` will add `x`, converted into a string representation, onto the end of the data that was already in the buffer. This command actually modifies the buffer, rather than making a copy, and that can be done efficiently. A long string can be built up in a *StringBuffer* using a sequence of `append()` commands. When the string is complete, the function `buffer.toString()` will return a copy of the string in the buffer as an ordinary value

of type *String*. The *StringBuffer* class is in the standard package `java.lang`, so you can use its simple name without importing it.

A number of useful classes are collected in the package `java.util`. For example, this package contains classes for working with collections of objects. We will encounter an example in Section 5.5, and we will study the collection classes extensively in Chapter 10. Another class in this package, `java.util.Date`, is used to represent times. When a `Date` object is constructed without parameters, the result represents the current date and time, so an easy way to display this information is:

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

Of course, since it is in the package `java.util`, in order to use the `Date` class in your program, you must make it available by importing it with one of the statements "`import java.util.Date;`" or "`import java.util.*;`" at the beginning of your program. (See Subsection 4.5.3 for a discussion of packages and `import`.)

I will also mention the class `java.util.Random`. An object belonging to this class is a *source* of random numbers (or, more precisely pseudorandom numbers). The standard function `Math.random()` uses one of these objects behind the scenes to generate its random numbers. An object of type `Random` can generate random integers, as well as random real numbers. If `randGen` is created with the command:

```
Random randGen = new Random();
```

and if `N` is a positive integer, then `randGen.nextInt(N)` generates a random integer in the range from `0` to `N-1`. For example, this makes it a little easier to roll a pair of dice. Instead of saying "`die1 = (int)(6*Math.random())+1;`", one can say "`die1 = randGen.nextInt(6)+1;`". (Since you also have to import the class `java.util.Random` and create the `Random` object, you might not agree that it is actually easier.) An object of type `Random` can also be used to generate so-called Gaussian distributed random real numbers.

The main point here, again, is that many problems have already been solved, and the solutions are available in Java's standard classes. If you are faced with a task that looks like it should be fairly common, it might be worth looking through a Java reference to see whether someone has already written a class that you can use.

5.3.2 Wrapper Classes and Autoboxing

We have already encountered the classes *Double* and *Integer* in Subsection 2.5.7. These classes contain the `static` methods `Double.parseDouble` and `Integer.parseInteger` that are used to convert strings to numerical values. We have also encountered the *Character* class in some examples, with static methods such as `Character.isLetter`, which can be used to test whether a given value of type `char` is a letter. There is a similar class for each of the other primitive types, *Long*, *Short*, *Byte*, *Float*, and *Boolean*. These classes are called ***wrapper classes***. Although they contain useful `static` members, they have another use as well: They are used for creating objects that represent primitive type values.

Remember that the primitive types are not classes, and values of primitive type are not objects. However, sometimes it's useful to treat a primitive value as if it were an object. You can't do that literally, but you can "wrap" the primitive type value in an object belonging to one of the wrapper classes.

For example, an object of type *Double* contains a single instance variable, of type **double**. The object is a wrapper for the **double** value. For example, you can create an object that wraps the **double** value `6.0221415e23` with

```
Double d = new Double(6.0221415e23);
```

The value of `d` contains the same information as the value of type **double**, but it is an object. If you want to retrieve the **double** value that is wrapped in the object, you can call the function `d.doubleValue()`. Similarly, you can wrap an **int** in an object of type *Integer*, a **boolean** value in an object of type *Boolean*, and so on. (As an example of where this would be useful, the collection classes that will be studied in Chapter 10 can only hold objects. If you want to add a primitive type value to a collection, it has to be put into a wrapper object first.)

Since Java 5.0, wrapper classes have been even easier to use. Java 5.0 introduced automatic conversion between a primitive type and the corresponding wrapper class. For example, if you use a value of type **int** in a context that requires an object of type *Integer*, the **int** will automatically be wrapped in an *Integer* object. For example, you can say

```
Integer answer = 42;
```

and the computer will silently read this as if it were

```
Integer answer = new Integer(42);
```

This is called ***autoboxing***. It works in the other direction, too. For example, if `d` refers to an object of type `Double`, you can use `d` in a numerical expression such as `2*d`. The **double** value inside `d` is automatically ***unboxed*** and multiplied by 2. Autoboxing and unboxing also apply to subroutine calls. For example, you can pass an actual parameter of type **int** to a subroutine that has a formal parameter of type *Integer*. In fact, autoboxing and unboxing make it possible in many circumstances to ignore the difference between primitive types and objects.

* * *

The wrapper classes contain a few other things that deserve to be mentioned. *Integer*, for example, contains constants `Integer.MIN_VALUE` and `Integer.MAX_VALUE`, which are equal to the largest and smallest possible values of type **int**, that is, to -2147483648 and 2147483647 respectively. It's certainly easier to remember the names than the numerical values. There are similar named constants in *Long*, *Short*, and *Byte*. *Double* and *Float* also have constants named `MIN_VALUE` and `MAX_VALUE`. `MAX_VALUE` still gives the largest number that can be represented in the given type, but `MIN_VALUE` represents the smallest possible **positive** value. For type **double**, `Double.MIN_VALUE` is 4.9 times 10^{-324}. Since **double** values have only a finite accuracy, they can't get arbitrarily close to zero. This is the closest they can get without actually being equal to zero.

The class *Double* deserves special mention, since **double**s are so much more complicated than integers. The encoding of real numbers into values of type **double** has room for a few special values that are not real numbers at all in the mathematical sense. These values are given by named constants in class *Double*: `Double.POSITIVE_INFINITY`, `Double.NEGATIVE_INFINITY`, and `Double.NaN`. The infinite values can occur as the values of certain mathematical expressions. For example, dividing a positive number by zero will give the result `Double.POSITIVE_INFINITY`. (It's even more complicated than this, actually, because the **double** type includes a value called "negative zero", written `-0.0`. Dividing a positive number by negative zero gives `Double.NEGATIVE_INFINITY`.) You also get `Double.POSITIVE_INFINITY` whenever the mathematical value of an expression is greater than `Double.MAX_VALUE`. For example, `1e200*1e200` is considered to be infinite. The value `Double.NaN` is even more interesting. "NaN" stands for ***Not a Number***, and it represents an undefined value such as the square root of a negative number or the result of dividing zero by zero. Because of the existence of `Double.NaN`, no mathematical operation on real numbers will ever throw an exception; it simply gives `Double.NaN` as the result.

You can test whether a value, `x`, of type **double** is infinite or undefined by calling the boolean-valued static functions `Double.isInfinite(x)` and `Double.isNaN(x)`. (It's especially important to use `Double.isNaN()` to test for undefined values, because `Double.NaN` has really weird behavior when used with relational operators such as `==`. In fact, the values of `x == Double.NaN` and `x != Double.NaN` are always **both false**—no matter what the value of `x` is—so you can't use these expressions to test whether `x` is `Double.NaN`.)

5.3.3 The class "Object"

We have already seen that one of the major features of object-oriented programming is the ability to create subclasses of a class. The subclass inherits all the properties or behaviors of the class, but can modify and add to what it inherits. In Section 5.5, you'll learn how to create subclasses. What you don't know yet is that **every** class in Java (with just one exception) is a subclass of some other class. If you create a class and don't explicitly make it a subclass of some other class, then it automatically becomes a subclass of the special class named *Object*. (*Object* is the one class that is not a subclass of any other class.)

Class *Object* defines several instance methods that are inherited by every other class. These methods can be used with any object whatsoever. I will mention just one of them here. You will encounter more of them later in the book.

The instance method `toString()` in class *Object* returns a value of type *String* that is supposed to be a string representation of the object. You've already used this method implicitly, any time you've printed out an object or concatenated an object onto a string. When you use an object in a context that requires a string, the object is automatically converted to type *String* by calling its `toString()` method.

The version of `toString` that is defined in *Object* just returns the name of the class that the object belongs to, concatenated with a code number called the ***hash code*** of the object; this is not very useful. When you create a class, you can write a new `toString()` method for it, which will replace the inherited version. For example, we might add the following method to any of the *PairOfDice* classes from the previous section:

```
/**
 * Return a String representation of a pair of dice, where die1
 * and die2 are instance variables containing the numbers that are
 * showing on the two dice.
 */
public String toString() {
   if (die1 == die2)
      return "double " + die1;
   else
      return die1 + " and " + die2;
}
```

If `dice` refers to a *PairOfDice* object, then `dice.toString()` will return strings such as "3 and 6", "5 and 1", and "double 2", depending on the numbers showing on the dice. This method would be used automatically to convert `dice` to type *String* in a statement such as

```
System.out.println( "The dice came up " + dice );
```

so this statement might output, "The dice came up 5 and 1" or "The dice came up double 2". You'll see another example of a `toString()` method in the next section.

5.3.4 Object-oriented Analysis and Design

Every programmer builds up a stock of techniques and expertise expressed as snippets of code that can be reused in new programs using the tried-and-true method of cut-and-paste: The old code is physically copied into the new program and then edited to customize it as necessary. The problem is that the editing is error-prone and time-consuming, and the whole enterprise is dependent on the programmer's ability to pull out that particular piece of code from last year's project that looks like it might be made to fit. (On the level of a corporation that wants to save money by not reinventing the wheel for each new project, just keeping track of all the old wheels becomes a major task.)

Well-designed classes are software components that can be reused without editing. A well-designed class is not carefully crafted to do a particular job in a particular program. Instead, it is crafted to model some particular type of object or a single coherent concept. Since objects and concepts can recur in many problems, a well-designed class is likely to be reusable without modification in a variety of projects.

Furthermore, in an object-oriented programming language, it is possible to make ***subclasses*** of an existing class. This makes classes even more reusable. If a class needs to be customized, a subclass can be created, and additions or modifications can be made in the subclass without making any changes to the original class. This can be done even if the programmer doesn't have access to the source code of the class and doesn't know any details of its internal, hidden implementation.

* * *

The *PairOfDice* class in the previous section is already an example of a generalized software component, although one that could certainly be improved. The class represents a single, coherent concept, "a pair of dice." The instance variables hold the data relevant to the state of the dice, that is, the number showing on each of the dice. The instance method represents the behavior of a pair of dice, that is, the ability to be rolled. This class would be reusable in many different programming projects.

On the other hand, the *Student* class from the previous section is not very reusable. It seems to be crafted to represent students in a particular course where the grade will be based on three tests. If there are more tests or quizzes or papers, it's useless. If there are two people in the class who have the same name, we are in trouble (one reason why numerical student ID's are often used). Admittedly, it's much more difficult to develop a general-purpose student class than a general-purpose pair-of-dice class. But this particular *Student* class is good mostly as an example in a programming textbook.

* * *

A large programming project goes through a number of stages, starting with ***specification*** of the problem to be solved, followed by ***analysis*** of the problem and ***design*** of a program to solve it. Then comes ***coding***, in which the program's design is expressed in some actual programming language. This is followed by ***testing*** and ***debugging*** of the program. After that comes a long period of ***maintenance***, which means fixing any new problems that are found in the program and modifying it to adapt it to changing requirements. Together, these stages form what is called the ***software life cycle***. (In the real world, the ideal of consecutive stages is seldom if ever achieved. During the analysis stage, it might turn out that the specifications are incomplete or inconsistent. A problem found during testing requires at least a brief return to the coding stage. If the problem is serious enough, it might even require a new design. Maintenance usually involves redoing some of the work from previous stages. . . .)

Large, complex programming projects are only likely to succeed if a careful, systematic approach is adopted during all stages of the software life cycle. The systematic approach to programming, using accepted principles of good design, is called ***software engineering***. The software engineer tries to efficiently construct programs that verifiably meet their specifications and that are easy to modify if necessary. There is a wide range of "methodologies" that can be applied to help in the systematic design of programs. (Most of these methodologies seem to involve drawing little boxes to represent program components, with labeled arrows to represent relationships among the boxes.)

We have been discussing object orientation in programming languages, which is relevant to the coding stage of program development. But there are also object-oriented methodologies for analysis and design. The question in this stage of the software life cycle is, How can one discover or invent the overall structure of a program? As an example of a rather simple object-oriented approach to analysis and design, consider this advice: Write down a description of the problem. Underline all the nouns in that description. The nouns should be considered as candidates for becoming classes or objects in the program design. Similarly, underline all the verbs. These are candidates for methods. This is your starting point. Further analysis might uncover the need for more classes and methods, and it might reveal that subclassing can be used to take advantage of similarities among classes.

This is perhaps a bit simple-minded, but the idea is clear and the general approach can be effective: Analyze the problem to discover the concepts that are involved, and create classes to represent those concepts. The design should arise from the problem itself, and you should end up with a program whose structure reflects the structure of the problem in a natural way.

5.4 Programming Example: Card, Hand, Deck

In this section, we look at some specific examples of object-oriented design in a domain that is simple enough that we have a chance of coming up with something reasonably reusable. Consider card games that are played with a standard deck of playing cards (a so-called "poker" deck, since it is used in the game of poker).

5.4.1 Designing the classes

In a typical card game, each player gets a hand of cards. The deck is shuffled and cards are dealt one at a time from the deck and added to the players' hands. In some games, cards can be removed from a hand, and new cards can be added. The game is won or lost depending on the value (ace, 2, ..., king) and suit (spades, diamonds, clubs, hearts) of the cards that a player receives. If we look for nouns in this description, there are several candidates for objects: game, player, hand, card, deck, value, and suit. Of these, the value and the suit of a card are simple values, and they might just be represented as instance variables in a *Card* object. In a complete program, the other five nouns might be represented by classes. But let's work on the ones that are most obviously reusable: card, hand, and deck.

If we look for verbs in the description of a card game, we see that we can shuffle a deck and deal a card from a deck. This gives use us two candidates for instance methods in a *Deck* class: `shuffle()` and `dealCard()`. Cards can be added to and removed from hands. This gives two candidates for instance methods in a *Hand* class: `addCard()` and `removeCard()`. Cards are relatively passive things, but we need to be able to determine their suits and values. We will discover more instance methods as we go along.

First, we'll design the deck class in detail. When a deck of cards is first created, it contains 52 cards in some standard order. The *Deck* class will need a constructor to create a new deck. The constructor needs no parameters because any new deck is the same as any other. There will be an instance method called `shuffle()` that will rearrange the 52 cards into a random order. The `dealCard()` instance method will get the next card from the deck. This will be a function with a return type of `Card`, since the caller needs to know what card is being dealt. It has no parameters—when you deal the next card from the deck, you don't provide any information to the deck; you just get the next card, whatever it is. What will happen if there are no more cards in the deck when its `dealCard()` method is called? It should probably be considered an error to try to deal a card from an empty deck, so the deck can throw an exception in that case. But this raises another question: How will the rest of the program know whether the deck is empty? Of course, the program could keep track of how many cards it has used. But the deck itself should know how many cards it has left, so the program should just be able to ask the deck object. We can make this possible by specifying another instance method, `cardsLeft()`, that returns the number of cards remaining in the deck. This leads to a full specification of all the subroutines in the *Deck* class:

Constructor and instance methods in class Deck:

```
/**
 * Constructor.  Create an unshuffled deck of cards.
 */
public Deck()

/**
 * Put all the used cards back into the deck,
 * and shuffle it into a random order.
 */
public void shuffle()

/**
 * As cards are dealt from the deck, the number of
 * cards left decreases.  This function returns the
 * number of cards that are still left in the deck.
 */
public int cardsLeft()

/**
 * Deals one card from the deck and returns it.
 * @throws IllegalStateException if no more cards are left.
 */
public Card dealCard()
```

This is everything you need to know in order to use the `Deck` class. Of course, it doesn't tell us how to write the class. This has been an exercise in design, not in coding. In fact, writing the class involves a programming technique, arrays, which will not be covered until Chapter 7. Nevertheless, you can look at the source code, *Deck.java*, if you want. Even though you won't understand the implementation, the Javadoc comments give you all the information that you need to understand the interface. With this information, you can use the class in your programs without understanding the implementation.

We can do a similar analysis for the *Hand* class. When a hand object is first created, it has no cards in it. An `addCard()` instance method will add a card to the hand. This method needs a parameter of type *Card* to specify which card is being added. For the `removeCard()`

method, a parameter is needed to specify which card to remove. But should we specify the card itself ("Remove the ace of spades"), or should we specify the card by its position in the hand ("Remove the third card in the hand")? Actually, we don't have to decide, since we can allow for both options. We'll have two `removeCard()` instance methods, one with a parameter of type *Card* specifying the card to be removed and one with a parameter of type **int** specifying the position of the card in the hand. (Remember that you can have two methods in a class with the same name, provided they have different numbers or types of parameters.) Since a hand can contain a variable number of cards, it's convenient to be able to ask a hand object how many cards it contains. So, we need an instance method `getCardCount()` that returns the number of cards in the hand. When I play cards, I like to arrange the cards in my hand so that cards of the same value are next to each other. Since this is a generally useful thing to be able to do, we can provide instance methods for sorting the cards in the hand. Here is a full specification for a reusable *Hand* class:

Constructor and instance methods in class Hand:

```
/**
 * Constructor. Create a Hand object that is initially empty.
 */
public Hand()

/**
 * Discard all cards from the hand, making the hand empty.
 */
public void clear()

/**
 * Add the card c to the hand.  c should be non-null.
 * @throws NullPointerException if c is null.
 */
public void addCard(Card c)

/**
 * If the specified card is in the hand, it is removed.
 */
public void removeCard(Card c)

/**
 * Remove the card in the specified position from the
 * hand.  Cards are numbered counting from zero.
 * @throws IllegalArgumentException if the specified
 *    position does not exist in the hand.
 */
public void removeCard(int position)

/**
 * Return the number of cards in the hand.
 */
public int getCardCount()

/**
 * Get the card from the hand in given position, where
 * positions are numbered starting from 0.
 * @throws IllegalArgumentException if the specified
 *    position does not exist in the hand.
```

```
     */
    public Card getCard(int position)

    /**
     * Sorts the cards in the hand so that cards of the same
     * suit are grouped together, and within a suit the cards
     * are sorted by value.
     */
    public void sortBySuit()

    /**
     * Sorts the cards in the hand so that cards are sorted into
     * order of increasing value.  Cards with the same value
     * are sorted by suit. Note that aces are considered
     * to have the lowest value.
     */
    public void sortByValue()
```

Again, you don't yet know enough to implement this class. But given the source code, *Hand.java*, you can use the class in your own programming projects.

5.4.2 The Card Class

We **have** covered enough material to write a *Card* class. The class will have a constructor that specifies the value and suit of the card that is being created. There are four suits, which can be represented by the integers 0, 1, 2, and 3. It would be tough to remember which number represents which suit, so I've defined named constants in the *Card* class to represent the four possibilities. For example, `Card.SPADES` is a constant that represents the suit, spades. (These constants are declared to be `public final static int`s. It might be better to use an enumerated type, but for now we will stick to integer-valued constants. I'll return to the question of using enumerated types in this example at the end of the chapter.) The possible values of a card are the numbers 1, 2, ..., 13, with 1 standing for an ace, 11 for a jack, 12 for a queen, and 13 for a king. Again, I've defined some named constants to represent the values of aces and face cards. (When you read the *Card* class, you'll see that I've also added support for Jokers.)

A *Card* object can be constructed knowing the value and the suit of the card. For example, we can call the constructor with statements such as:

```
card1 = new Card( Card.ACE, Card.SPADES );  // Construct ace of spades.
card2 = new Card( 10, Card.DIAMONDS );   // Construct 10 of diamonds.
card3 = new Card( v, s );  // This is OK, as long as v and s
                           //               are integer expressions.
```

A *Card* object needs instance variables to represent its value and suit. I've made these `private` so that they cannot be changed from outside the class, and I've provided getter methods `getSuit()` and `getValue()` so that it will be possible to discover the suit and value from outside the class. The instance variables are initialized in the constructor, and are never changed after that. In fact, I've declared the instance variables `suit` and `value` to be `final`, since they are never changed after they are initialized. (An instance variable can be declared `final` provided it is either given an initial value in its declaration or is initialized in every constructor in the class.)

Finally, I've added a few convenience methods to the class to make it easier to print out cards in a human-readable form. For example, I want to be able to print out the suit of a

card as the word "Diamonds", rather than as the meaningless code number 2, which is used in the class to represent diamonds. Since this is something that I'll probably have to do in many programs, it makes sense to include support for it in the class. So, I've provided instance methods `getSuitAsString()` and `getValueAsString()` to return string representations of the suit and value of a card. Finally, I've defined the instance method `toString()` to return a string with both the value and suit, such as "Queen of Hearts". Recall that this method will be used automatically whenever a *Card* needs to be converted into a *String*, such as when the card is concatenated onto a string with the + operator. Thus, the statement

```
System.out.println( "Your card is the " + card );
```

is equivalent to

```
System.out.println( "Your card is the " + card.toString() );
```

If the card is the queen of hearts, either of these will print out "Your card is the Queen of Hearts".

Here is the complete *Card* class. It is general enough to be highly reusable, so the work that went into designing, writing, and testing it pays off handsomely in the long run.

```
/**
 * An object of type Card represents a playing card from a
 * standard Poker deck, including Jokers.  The card has a suit, which
 * can be spades, hearts, diamonds, clubs, or joker.  A spade, heart,
 * diamond, or club has one of the 13 values: ace, 2, 3, 4, 5, 6, 7,
 * 8, 9, 10, jack, queen, or king.  Note that "ace" is considered to be
 * the smallest value.  A joker can also have an associated value;
 * this value can be anything and can be used to keep track of several
 * different jokers.
 */

public class Card {

   public final static int SPADES = 0;   // Codes for the 4 suits, plus Joker.
   public final static int HEARTS = 1;
   public final static int DIAMONDS = 2;
   public final static int CLUBS = 3;
   public final static int JOKER = 4;

   public final static int ACE = 1;      // Codes for the non-numeric cards.
   public final static int JACK = 11;    //   Cards 2 through 10 have their
   public final static int QUEEN = 12;   //   numerical values for their codes.
   public final static int KING = 13;

   /**
    * This card's suit, one of the constants SPADES, HEARTS, DIAMONDS,
    * CLUBS, or JOKER.  The suit cannot be changed after the card is
    * constructed.
    */
   private final int suit;

   /**
    * The card's value.  For a normal card, this is one of the values
    * 1 through 13, with 1 representing ACE.  For a JOKER, the value
    * can be anything.  The value cannot be changed after the card
    * is constructed.
    */
```

```
private final int value;

/**
 * Creates a Joker, with 1 as the associated value.  (Note that
 * "new Card()" is equivalent to "new Card(1,Card.JOKER)".)
 */
public Card() {
   suit = JOKER;
   value = 1;
}

/**
 * Creates a card with a specified suit and value.
 * @param theValue the value of the new card.  For a regular card (non-joker),
 * the value must be in the range 1 through 13, with 1 representing an Ace.
 * You can use the constants Card.ACE, Card.JACK, Card.QUEEN, and Card.KING.
 * For a Joker, the value can be anything.
 * @param theSuit the suit of the new card.  This must be one of the values
 * Card.SPADES, Card.HEARTS, Card.DIAMONDS, Card.CLUBS, or Card.JOKER.
 * @throws IllegalArgumentException if the parameter values are not in the
 * permissible ranges
 */
public Card(int theValue, int theSuit) {
   if (theSuit != SPADES && theSuit != HEARTS && theSuit != DIAMONDS &&
         theSuit != CLUBS && theSuit != JOKER)
      throw new IllegalArgumentException("Illegal playing card suit");
   if (theSuit != JOKER && (theValue < 1 || theValue > 13))
      throw new IllegalArgumentException("Illegal playing card value");
   value = theValue;
   suit = theSuit;
}

/**
 * Returns the suit of this card.
 * @returns the suit, which is one of the constants Card.SPADES,
 * Card.HEARTS, Card.DIAMONDS, Card.CLUBS, or Card.JOKER
 */
public int getSuit() {
   return suit;
}

/**
 * Returns the value of this card.
 * @return the value, which is one of the numbers 1 through 13, inclusive for
 * a regular card, and which can be any value for a Joker.
 */
public int getValue() {
   return value;
}

/**
 * Returns a String representation of the card's suit.
 * @return one of the strings "Spades", "Hearts", "Diamonds", "Clubs"
 * or "Joker".
 */
public String getSuitAsString() {
```

```
   switch ( suit ) {
   case SPADES:   return "Spades";
   case HEARTS:   return "Hearts";
   case DIAMONDS: return "Diamonds";
   case CLUBS:    return "Clubs";
   default:       return "Joker";
   }
}

/**
 * Returns a String representation of the card's value.
 * @return for a regular card, one of the strings "Ace", "2",
 * "3", ..., "10", "Jack", "Queen", or "King".  For a Joker, the
 * string is always numerical.
 */
public String getValueAsString() {
   if (suit == JOKER)
      return "" + value;
   else {
      switch ( value ) {
      case 1:   return "Ace";
      case 2:   return "2";
      case 3:   return "3";
      case 4:   return "4";
      case 5:   return "5";
      case 6:   return "6";
      case 7:   return "7";
      case 8:   return "8";
      case 9:   return "9";
      case 10:  return "10";
      case 11:  return "Jack";
      case 12:  return "Queen";
      default:  return "King";
      }
   }
}

/**
 * Returns a string representation of this card, including both
 * its suit and its value (except that for a Joker with value 1,
 * the return value is just "Joker").  Sample return values
 * are: "Queen of Hearts", "10 of Diamonds", "Ace of Spades",
 * "Joker", "Joker #2"
 */
public String toString() {
   if (suit == JOKER) {
      if (value == 1)
         return "Joker";
      else
         return "Joker #" + value;
   }
   else
      return getValueAsString() + " of " + getSuitAsString();
}
```

```
} // end class Card
```

5.4.3 Example: A Simple Card Game

I will finish this section by presenting a complete program that uses the *Card* and *Deck* classes. The program lets the user play a very simple card game called HighLow. A deck of cards is shuffled, and one card is dealt from the deck and shown to the user. The user predicts whether the next card from the deck will be higher or lower than the current card. If the user predicts correctly, then the next card from the deck becomes the current card, and the user makes another prediction. This continues until the user makes an incorrect prediction. The number of correct predictions is the user's score.

My program has a static method that plays one game of HighLow. This method has a return value that represents the user's score in the game. The `main()` routine lets the user play several games of HighLow. At the end, it reports the user's average score.

I won't go through the development of the algorithms used in this program, but I encourage you to read it carefully and make sure that you understand how it works. Note in particular that the subroutine that plays one game of HighLow returns the user's score in the game as its return value. This gets the score back to the main program, where it is needed. Here is the program:

```
/**
 * This program lets the user play HighLow, a simple card game
 * that is described in the output statements at the beginning of
 * the main() routine.  After the user plays several games,
 * the user's average score is reported.
 */

public class HighLow {


   public static void main(String[] args) {

      System.out.println("This program lets you play the simple card game,");
      System.out.println("HighLow.  A card is dealt from a deck of cards.");
      System.out.println("You have to predict whether the next card will be");
      System.out.println("higher or lower.  Your score in the game is the");
      System.out.println("number of correct predictions you make before");
      System.out.println("you guess wrong.");
      System.out.println();

      int gamesPlayed = 0;     // Number of games user has played.
      int sumOfScores = 0;     // The sum of all the scores from
                               //      all the games played.
      double averageScore;     // Average score, computed by dividing
                               //      sumOfScores by gamesPlayed.
      boolean playAgain;       // Record user's response when user is
                               //   asked whether he wants to play
                               //   another game.

      do {
         int scoreThisGame;        // Score for one game.
         scoreThisGame = play();   // Play the game and get the score.
         sumOfScores += scoreThisGame;
         gamesPlayed++;
```

```
      TextIO.put("Play again? ");
      playAgain = TextIO.getlnBoolean();
   } while (playAgain);

   averageScore = ((double)sumOfScores) / gamesPlayed;

   System.out.println();
   System.out.println("You played " + gamesPlayed + " games.");
   System.out.printf("Your average score was %1.3f.\n", averageScore);

}  // end main()


/**
 * Lets the user play one game of HighLow, and returns the
 * user's score on that game.  The score is the number of
 * correct guesses that the user makes.
 */
private static int play() {

   Deck deck = new Deck();  // Get a new deck of cards, and
                            //   store a reference to it in
                            //   the variable, deck.

   Card currentCard;  // The current card, which the user sees.

   Card nextCard;   // The next card in the deck.  The user tries
                    //    to predict whether this is higher or lower
                    //    than the current card.

   int correctGuesses ;  // The number of correct predictions the
                         //   user has made.  At the end of the game,
                         //   this will be the user's score.

   char guess;   // The user's guess.  'H' if the user predicts that
                 //   the next card will be higher, 'L' if the user
                 //   predicts that it will be lower.

   deck.shuffle();  // Shuffle the deck into a random order before
                    //    starting the game.

   correctGuesses = 0;
   currentCard = deck.dealCard();
   TextIO.putln("The first card is the " + currentCard);

   while (true) {  // Loop ends when user's prediction is wrong.

      /* Get the user's prediction, 'H' or 'L' (or 'h' or 'l'). */

      TextIO.put("Will the next card be higher (H) or lower (L)?  ");
      do {
          guess = TextIO.getlnChar();
          guess = Character.toUpperCase(guess);
          if (guess != 'H' && guess != 'L')
             TextIO.put("Please respond with H or L:  ");
      } while (guess != 'H' && guess != 'L');

      /* Get the next card and show it to the user. */

      nextCard = deck.dealCard();
      TextIO.putln("The next card is " + nextCard);
```

```
            /* Check the user's prediction. */

            if (nextCard.getValue() == currentCard.getValue()) {
               TextIO.putln("The value is the same as the previous card.");
               TextIO.putln("You lose on ties.  Sorry!");
               break;  // End the game.
            }
            else if (nextCard.getValue() > currentCard.getValue()) {
               if (guess == 'H') {
                   TextIO.putln("Your prediction was correct.");
                   correctGuesses++;
               }
               else {
                   TextIO.putln("Your prediction was incorrect.");
                   break;  // End the game.
               }
            }
            else {  // nextCard is lower
               if (guess == 'L') {
                   TextIO.putln("Your prediction was correct.");
                   correctGuesses++;
               }
               else {
                   TextIO.putln("Your prediction was incorrect.");
                   break;  // End the game.
               }
            }

            /* To set up for the next iteration of the loop, the nextCard
               becomes the currentCard, since the currentCard has to be
               the card that the user sees, and the nextCard will be
               set to the next card in the deck after the user makes
               his prediction.  */

            currentCard = nextCard;
            TextIO.putln();
            TextIO.putln("The card is " + currentCard);

         } // end of while loop

         TextIO.putln();
         TextIO.putln("The game is over.");
         TextIO.putln("You made " + correctGuesses
                                          + " correct predictions.");
         TextIO.putln();

         return correctGuesses;

      }  // end play()

   } // end class
```

5.5 Inheritance, Polymorphism, and Abstract Classes

A CLASS REPRESENTS A SET OF OBJECTS which share the same structure and behaviors. The class determines the structure of objects by specifying variables that are contained in each instance of the class, and it determines behavior by providing the instance methods that express the behavior of the objects. This is a powerful idea. However, something like this can be done in most programming languages. The central new idea in object-oriented programming—the idea that really distinguishes it from traditional programming—is to allow classes to express the similarities among objects that share **some**, but not all, of their structure and behavior. Such similarities can be expressed using ***inheritance*** and ***polymorphism***.

5.5.1 Extending Existing Classes

The topics covered in later subsections of this section are relatively advanced aspects of object-oriented programming. Any programmer should know what is meant by subclass, inheritance, and polymorphism. However, it will probably be a while before you actually do anything with inheritance except for extending classes that already exist. In the first part of this section, we look at how that is done.

In day-to-day programming, especially for programmers who are just beginning to work with objects, subclassing is used mainly in one situation: There is an existing class that can be adapted with a few changes or additions. This is much more common than designing groups of classes and subclasses from scratch. The existing class can be ***extended*** to make a subclass. The syntax for this is

```
public class ⟨subclass-name⟩ extends ⟨existing-class-name⟩ {
   .
   .   // Changes and additions.
   .
}
```

As an example, suppose you want to write a program that plays the card game, Blackjack. You can use the *Card*, *Hand*, and *Deck* classes developed in Section 5.4. However, a hand in the game of Blackjack is a little different from a hand of cards in general, since it must be possible to compute the "value" of a Blackjack hand according to the rules of the game. The rules are as follows: The value of a hand is obtained by adding up the values of the cards in the hand. The value of a numeric card such as a three or a ten is its numerical value. The value of a Jack, Queen, or King is 10. The value of an Ace can be either 1 or 11. An Ace should be counted as 11 unless doing so would put the total value of the hand over 21. Note that this means that the second, third, or fourth Ace in the hand will always be counted as 1.

One way to handle this is to extend the existing *Hand* class by adding a method that computes the Blackjack value of the hand. Here's the definition of such a class:

```
public class BlackjackHand extends Hand {

    /**
     * Computes and returns the value of this hand in the game
     * of Blackjack.
     */
    public int getBlackjackValue() {

        int val;      // The value computed for the hand.
        boolean ace;  // This will be set to true if the
```

```
                //   hand contains an ace.
        int cards;    // Number of cards in the hand.

        val = 0;
        ace = false;
        cards = getCardCount();  // (method defined in class Hand.)

        for ( int i = 0;  i < cards;  i++ ) {
                // Add the value of the i-th card in the hand.
            Card card;    // The i-th card;
            int cardVal;  // The blackjack value of the i-th card.
            card = getCard(i);
            cardVal = card.getValue();  // The normal value, 1 to 13.
            if (cardVal > 10) {
                cardVal = 10;   // For a Jack, Queen, or King.
            }
            if (cardVal == 1) {
                ace = true;     // There is at least one ace.
            }
            val = val + cardVal;
         }

         // Now, val is the value of the hand, counting any ace as 1.
         // If there is an ace, and if changing its value from 1 to
         // 11 would leave the score less than or equal to 21,
         // then do so by adding the extra 10 points to val.

         if ( ace == true  &&  val + 10 <= 21 )
             val = val + 10;

         return val;

    }  // end getBlackjackValue()

} // end class BlackjackHand
```

Since *BlackjackHand* is a subclass of *Hand*, an object of type *BlackjackHand* contains all the instance variables and instance methods defined in *Hand*, plus the new instance method named `getBlackjackValue()`. For example, if `bjh` is a variable of type `BlackjackHand`, then the following are all legal: `bjh.getCardCount()`, `bjh.removeCard(0)`, and `bjh.getBlackjackValue()`. The first two methods are defined in *Hand*, but are inherited by *BlackjackHand*.

Variables and methods from the *Hand* class are inherited by *BlackjackHand*, and they can be used in the definition of *BlackjackHand* just as if they were actually defined in that class (except for any that are declared to be `private`, which prevents access even by subclasses). The statement "`cards = getCardCount();`" in the above definition of `getBlackjackValue()` calls the instance method `getCardCount()`, which was defined in *Hand*.

Extending existing classes is an easy way to build on previous work. We'll see that many standard classes have been written specifically to be used as the basis for making subclasses.

* * *

Access modifiers such as `public` and `private` are used to control access to members of a class. There is one more access modifier, ***protected***, that comes into the picture when subclasses are taken into consideration. When `protected` is applied as an access modifier to a method or member variable in a class, that member can be used in subclasses—direct or indirect—of the

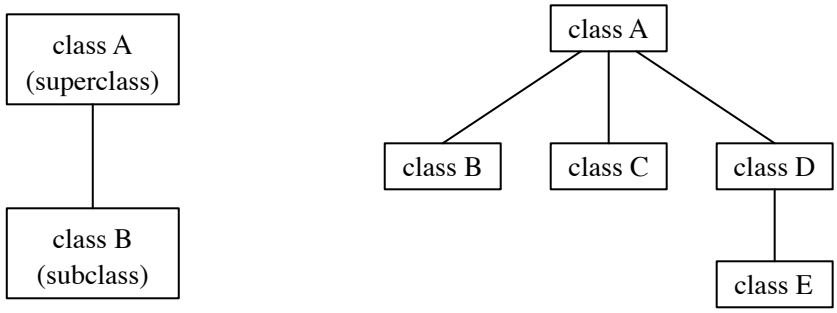
class in which it is defined, but it cannot be used in non-subclasses. (There is an exception: A `protected` member can also be accessed by any class in the same package as the class that contains the protected member. Recall that using no access modifier makes a member accessible to classes in the same package, and nowhere else. Using the `protected` modifier is strictly more liberal than using no modifier at all: It allows access from classes in the same package and from **subclasses** that are not in the same package.)

When you declare a method or member variable to be `protected`, you are saying that it is part of the implementation of the class, rather than part of the public interface of the class. However, you are allowing subclasses to use and modify that part of the implementation.

For example, consider a *PairOfDice* class that has instance variables `die1` and `die2` to represent the numbers appearing on the two dice. We could make those variables `private` to make it impossible to change their values from outside the class, while still allowing read access through getter methods. However, if we think it possible that *PairOfDice* will be used to create subclasses, we might want to make it possible for subclasses to change the numbers on the dice. For example, a *GraphicalDice* subclass that draws the dice might want to change the numbers at other times besides when the dice are rolled. In that case, we could make `die1` and `die2` `protected`, which would allow the subclass to change their values without making them public to the rest of the world. (An even better idea would be to define `protected` setter methods for the variables. A setter method could, for example, ensure that the value that is being assigned to the variable is in the legal range 1 through 6.)

5.5.2 Inheritance and Class Hierarchy

The term ***inheritance*** refers to the fact that one class can inherit part or all of its structure and behavior from another class. The class that does the inheriting is said to be a ***subclass*** of the class from which it inherits. If class B is a subclass of class A, we also say that class A is a ***superclass*** of class B. (Sometimes the terms ***derived class*** and ***base class*** are used instead of subclass and superclass; this is the common terminology in C++.) A subclass can add to the structure and behavior that it inherits. It can also replace or modify inherited behavior (though not inherited structure). The relationship between subclass and superclass is sometimes shown by a diagram in which the subclass is shown below, and connected to, its superclass, as shown on the left below:



In Java, to create a class named "B" as a subclass of a class named "A", you would write

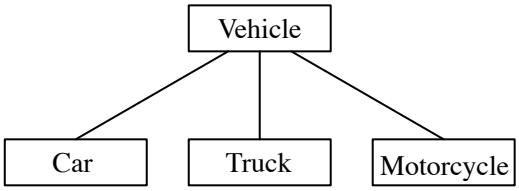
```
class B extends A {
    .
    .  // additions to, and modifications of,
```

```
    .  // stuff inherited from class A
    .
}
```

Several classes can be declared as subclasses of the same superclass. The subclasses, which might be referred to as "sibling classes," share some structures and behaviors—namely, the ones they inherit from their common superclass. The superclass expresses these shared structures and behaviors. In the diagram shown on the right above, classes B, C, and D are sibling classes. Inheritance can also extend over several "generations" of classes. This is shown in the diagram, where class E is a subclass of class D which is itself a subclass of class A. In this case, class E is considered to be a subclass of class A, even though it is not a direct subclass. This whole set of classes forms a small ***class hierarchy***.

5.5.3 Example: Vehicles

Let's look at an example. Suppose that a program has to deal with motor vehicles, including cars, trucks, and motorcycles. (This might be a program used by a Department of Motor Vehicles to keep track of registrations.) The program could use a class named *Vehicle* to represent all types of vehicles. Since cars, trucks, and motorcycles are types of vehicles, they would be represented by subclasses of the *Vehicle* class, as shown in this class hierarchy diagram:



The *Vehicle* class would include instance variables such as `registrationNumber` and `owner` and instance methods such as `transferOwnership()`. These are variables and methods common to all vehicles. The three subclasses of *Vehicle*—*Car*, *Truck*, and *Motorcycle*—could then be used to hold variables and methods specific to particular types of vehicles. The *Car* class might add an instance variable `numberOfDoors`, the *Truck* class might have `numberOfAxles`, and the *Motorcycle* class could have a boolean variable `hasSidecar`. (Well, it could in theory at least, even if it might give a chuckle to the people at the Department of Motor Vehicles.) The declarations of these classes in a Java program would look, in outline, like this (although in practice, they would probably be `public` classes, defined in separate files):

```
class Vehicle {
   int registrationNumber;
   Person owner;  // (Assuming that a Person class has been defined!)
   void transferOwnership(Person newOwner) {
       . . .
   }
   . . .
}

class Car extends Vehicle {
   int numberOfDoors;
   . . .
```

```
}

class Truck extends Vehicle {
   int numberOfAxles;
   . . .
}

class Motorcycle extends Vehicle {
   boolean hasSidecar;
   . . .
}
```

Suppose that `myCar` is a variable of type *Car* that has been declared and initialized with the statement

```
Car myCar = new Car();
```

Given this declaration, a program could refer to `myCar.numberOfDoors`, since `numberOfDoors` is an instance variable in the class `Car`. But since class *Car* extends class *Vehicle*, a car also has all the structure and behavior of a vehicle. This means that `myCar.registrationNumber`, `myCar.owner`, and `myCar.transferOwnership()` also exist.

Now, in the real world, cars, trucks, and motorcycles are in fact vehicles. The same is true in a program. That is, an object of type *Car* or *Truck* or *Motorcycle* is automatically an object of type *Vehicle* too. This brings us to the following Important Fact:

A variable that can hold a reference to an object of class A can also hold a reference to an object belonging to any subclass of A.

The practical effect of this in our example is that an object of type *Car* can be assigned to a variable of type *Vehicle*. That is, it would be legal to say

```
Vehicle myVehicle = myCar;
```

or even

```
Vehicle myVehicle = new Car();
```

After either of these statements, the variable `myVehicle` holds a reference to a *Vehicle* object that happens to be an instance of the subclass, *Car*. The object "remembers" that it is in fact a *Car*, and not **just** a *Vehicle*. Information about the actual class of an object is stored as part of that object. It is even possible to test whether a given object belongs to a given class, using the `instanceof` operator. The test:

```
if (myVehicle instanceof Car) ...
```

determines whether the object referred to by `myVehicle` is in fact a car.

On the other hand, the assignment statement

```
myCar = myVehicle;
```

would be illegal because `myVehicle` could potentially refer to other types of vehicles that are not cars. This is similar to a problem we saw previously in Subsection 2.5.6: The computer will not allow you to assign an **int** value to a variable of type **short**, because not every **int** is a **short**. Similarly, it will not allow you to assign a value of type *Vehicle* to a variable of type *Car* because not every vehicle is a car. As in the case of **int**s and **short**s, the solution here is to use type-casting. If, for some reason, you happen to know that `myVehicle` does in fact refer to a *Car*, you can use the type cast `(Car)myVehicle` to tell the computer to treat `myVehicle` as if it were actually of type *Car*. So, you could say

```
myCar = (Car)myVehicle;
```

and you could even refer to `((Car)myVehicle).numberOfDoors`. (The parentheses are necessary because of precedence. The "." has higher precedence than the type-cast, so `(Car)myVehicle.numberOfDoors` would try to type-cast the **int** `myVehicle.numberOfDoors` into a *Vehicle*, which is impossible.)

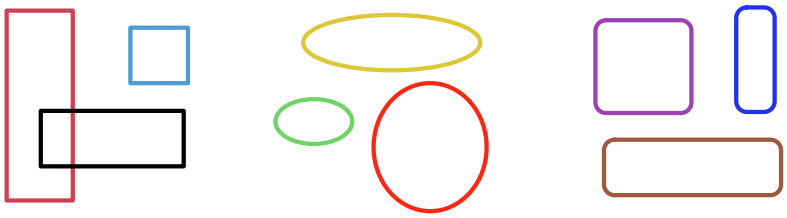
As an example of how this could be used in a program, suppose that you want to print out relevant data about the *Vehicle* referred to by `myVehicle`. If it's a *car*, you will want to print out the car's `numberOfDoors`, but you can't say `myVehicle.numberOfDoors`, since there is no `numberOfDoors` in the *Vehicle* class. But you could say:

```
System.out.println("Vehicle Data:");
System.out.println("Registration number:  "
                              + myVehicle.registrationNumber);
if (myVehicle instanceof Car) {
   System.out.println("Type of vehicle:  Car");
   Car c;
   c = (Car)myVehicle;  // Type-cast to get access to numberOfDoors!
   System.out.println("Number of doors:  " + c.numberOfDoors);
}
else if (myVehicle instanceof Truck) {
   System.out.println("Type of vehicle:  Truck");
   Truck t;
   t = (Truck)myVehicle;  // Type-cast to get access to numberOfAxles!
   System.out.println("Number of axles:  " + t.numberOfAxles);
}
else if (myVehicle instanceof Motorcycle) {
   System.out.println("Type of vehicle:  Motorcycle");
   Motorcycle m;
   m = (Motorcycle)myVehicle;  // Type-cast to get access to hasSidecar!
   System.out.println("Has a sidecar:    " + m.hasSidecar);
}
```

Note that for object types, when the computer executes a program, it checks whether type-casts are valid. So, for example, if `myVehicle` refers to an object of type *Truck*, then the type cast `(Car)myVehicle` would be an error. When this happens, an exception of type *ClassCastException* is thrown. This check is done at run time, not compile time, because the actual type of the object referred to by `myVehicle` is not known when the program is compiled.

5.5.4 Polymorphism

As another example, consider a program that deals with shapes drawn on the screen. Let's say that the shapes include rectangles, ovals, and roundrects of various colors. (A "roundrect" is just a rectangle with rounded corners.)



Rectangles *Ovals* *RoundRects*

Three classes, *Rectangle*, *Oval*, and *RoundRect*, could be used to represent the three types of shapes. These three classes would have a common superclass, *Shape*, to represent features that all three shapes have in common. The *Shape* class could include instance variables to represent the color, position, and size of a shape, and it could include instance methods for changing the color, position, and size. Changing the color, for example, might involve changing the value of an instance variable, and then redrawing the shape in its new color:

```
class Shape {

    Color color;   // Color of the shape.  (Recall that class Color
                   // is defined in package java.awt.  Assume
                   // that this class has been imported.)

    void setColor(Color newColor) {
          // Method to change the color of the shape.
       color = newColor; // change value of instance variable
       redraw(); // redraw shape, which will appear in new color
    }

    void redraw() {
          // method for drawing the shape
       ? ? ?  // what commands should go here?
    }

    . . .          // more instance variables and methods

} // end of class Shape
```

Now, you might see a problem here with the method `redraw()`. The problem is that each different type of shape is drawn differently. The method `setColor()` can be called for any type of shape. How does the computer know which shape to draw when it executes the `redraw()`? Informally, we can answer the question like this: The computer executes `redraw()` by asking the shape to redraw **itself**. Every shape object knows what it has to do to redraw itself.

In practice, this means that each of the specific shape classes has its own `redraw()` method:

```
class Rectangle extends Shape {
   void redraw() {
      . . .  // commands for drawing a rectangle
   }
   . . . // possibly, more methods and variables
}

class Oval extends Shape {
   void redraw() {
```

```
        . . .  // commands for drawing an oval
    }
    . . . // possibly, more methods and variables
}

class RoundRect extends Shape {
    void redraw() {
        . . .  // commands for drawing a rounded rectangle
    }
    . . . // possibly, more methods and variables
}
```

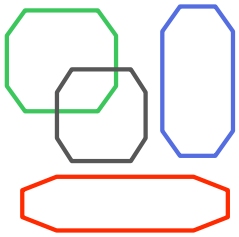
If `oneShape` is a variable of type *Shape*, it could refer to an object of any of the types *Rectangle*, *Oval*, or *RoundRect*. As a program executes, and the value of `oneShape` changes, it could even refer to objects of different types at different times! Whenever the statement

```
oneShape.redraw();
```

is executed, the redraw method that is actually called is the one appropriate for the type of object to which `oneShape` actually refers. There may be no way of telling, from looking at the text of the program, what shape this statement will draw, since it depends on the value that `oneShape` happens to have when the program is executed. Even more is true. Suppose the statement is in a loop and gets executed many times. If the value of `oneShape` changes as the loop is executed, it is possible that the very same statement "`oneShape.redraw();`" will call different methods and draw different shapes as it is executed over and over. We say that the `redraw()` method is ***polymorphic***. A method is polymorphic if the action performed by the method depends on the actual type of the object to which the method is applied. Polymorphism is one of the major distinguishing features of object-oriented programming.

Perhaps this becomes more understandable if we change our terminology a bit: In object-oriented programming, calling a method is often referred to as sending a ***message*** to an object. The object responds to the message by executing the appropriate method. The statement "`oneShape.redraw();`" is a message to the object referred to by `oneShape`. Since that object knows what type of object it is, it knows how it should respond to the message. From this point of view, the computer always executes "`oneShape.redraw();`" in the same way: by sending a message. The response to the message depends, naturally, on who receives it. From this point of view, objects are active entities that send and receive messages, and polymorphism is a natural, even necessary, part of this view. Polymorphism just means that different objects can respond to the same message in different ways.

One of the most beautiful things about polymorphism is that it lets code that you write do things that you didn't even conceive of, at the time you wrote it. Suppose that I decide to add beveled rectangles to the types of shapes my program can deal with. A beveled rectangle has a triangle cut off each corner:



BeveledRects

To implement beveled rectangles, I can write a new subclass, *BeveledRect*, of class *Shape* and give it its own `redraw()` method. Automatically, code that I wrote previously—such as the statement `oneShape.redraw()`—can now suddenly start drawing beveled rectangles, even though the beveled rectangle class didn't exist when I wrote the statement!

In the statement "`oneShape.redraw();`", the `redraw` message is sent to the object `oneShape`. Look back at the method in the *Shape* class for changing the color of a shape:

```
void setColor(Color newColor) {
   color = newColor; // change value of instance variable
   redraw(); // redraw shape, which will appear in new color
}
```

A `redraw` message is sent here, but which object is it sent to? Well, the `setColor` method is itself a message that was sent to some object. The answer is that the `redraw` message is sent to that **same object**, the one that received the `setColor` message. If that object is a rectangle, then it contains a `redraw()` method for drawing rectangles, and that is the one that is executed. If the object is an oval, then it is the `redraw()` method from the *Oval* class. This is what you should expect, but it means that the "`redraw();`" statement in the `setColor()` method does **not** necessarily call the `redraw()` method in the *Shape* class! The `redraw()` method that is executed could be in any subclass of *Shape*. This is just another case of polymorphism.

Again, this is not a real surprise if you think about it in the right way. Remember that an instance method is always contained in an object. The class only contains the source code for the method. When a *Rectangle* object is created, it contains a `redraw()` method. The source code for that method is in the *Rectangle* class. The object also contains a `setColor()` method. Since the *Rectangle* class does not define a `setColor()` method, the **source code** for the rectangle's `setColor()` method comes from the superclass, *Shape*, but the **method itself** is in the object of type *Rectangle*. Even though the source codes for the two methods are in different classes, the methods themselves are part of the same object. When the rectangle's `setColor()` method is executed and calls `redraw()`, the `redraw()` method that is executed is the one in the same object.

5.5.5 Abstract Classes

Whenever a *Rectangle*, *Oval*, or *RoundRect* object has to draw itself, it is the `redraw()` method in the appropriate class that is executed. This leaves open the question, What does the `redraw()` method in the *Shape* class do? How should it be defined?

The answer may be surprising: We should leave it blank! The fact is that the class *Shape* represents the abstract idea of a shape, and there is no way to draw such a thing. Only particular, concrete shapes like rectangles and ovals can be drawn. So, why should there

even be a `redraw()` method in the *Shape* class? Well, it has to be there, or it would be illegal to call it in the `setColor()` method of the *Shape* class, and it would be illegal to write "`oneShape.redraw();`". The compiler would complain that `oneShape` is a variable of type *Shape* and there's no `redraw()` method in the *Shape* class.

Nevertheless the version of `redraw()` in the *Shape* class itself will never actually be called. In fact, if you think about it, there can never be any reason to construct an actual object of type *Shape*! You can have **variables** of type *Shape*, but the objects they refer to will always belong to one of the subclasses of *Shape*. We say that *Shape* is an ***abstract class***. An abstract class is one that is not used to construct objects, but only as a basis for making subclasses. An abstract class exists **only** to express the common properties of all its subclasses. A class that is not abstract is said to be ***concrete***. You can create objects belonging to a concrete class, but not to an abstract class. A variable whose type is given by an abstract class can only refer to objects that belong to concrete subclasses of the abstract class.

Similarly, we say that the `redraw()` method in class *Shape* is an ***abstract method***, since it is never meant to be called. In fact, there is nothing for it to do—any actual redrawing is done by `redraw()` methods in the subclasses of *Shape*. The `redraw()` method in *Shape* has to be there. But it is there only to tell the computer that **all** `Shapes` understand the `redraw` message. As an abstract method, it exists merely to specify the common interface of all the actual, concrete versions of `redraw()` in the subclasses. There is no reason for the abstract `redraw()` in class *Shape* to contain any code at all.

Shape and its `redraw()` method are semantically abstract. You can also tell the computer, syntactically, that they are abstract by adding the modifier "`abstract`" to their definitions. For an abstract method, the block of code that gives the implementation of an ordinary method is replaced by a semicolon. An implementation must then be provided for the abstract method in any concrete subclass of the abstract class. Here's what the *Shape* class would look like as an abstract class:

```
public abstract class Shape {

    Color color;   // color of shape.

    void setColor(Color newColor) {
          // method to change the color of the shape
       color = newColor; // change value of instance variable
       redraw(); // redraw shape, which will appear in new color
    }

    abstract void redraw();
          // abstract method---must be defined in
          // concrete subclasses

    . . .  // more instance variables and methods

} // end of class Shape
```

Once you have declared the class to be `abstract`, it becomes illegal to try to create actual objects of type *Shape*, and the computer will report a syntax error if you try to do so.

Note, by the way, that the *Vehicle* class discussed above would probably also be an abstract class. There is no way to own a vehicle as such—the actual vehicle has to be a car or a truck or a motorcycle, or some other "concrete" type of vehicle.

* * *

Recall from Subsection 5.3.3 that a class that is not explicitly declared to be a subclass of some other class is automatically made a subclass of the standard class *Object*. That is, a class declaration with no "`extends`" part such as

```
public class myClass { . . .
```

is exactly equivalent to

```
public class myClass extends Object { . . .
```

This means that class *Object* is at the top of a huge class hierarchy that includes every other class. (Semantically, *Object* is an abstract class, in fact the most abstract class of all. Curiously, however, it is not declared to be `abstract` syntactically, which means that you can create objects of type *Object*. What you would do with them, however, I have no idea.)

Since every class is a subclass of *Object*, a variable of type *Object* can refer to any object whatsoever, of any type. Java has several standard data structures that are designed to hold *Objects*, but since every object is an instance of class *Object*, these data structures can actually hold any object whatsoever. One example is the "ArrayList" data structure, which is defined by the class *ArrayList* in the package `java.util`. (*ArrayList* is discussed more fully in Section 7.3.) An *ArrayList* is simply a list of *Objects*. This class is very convenient, because an *ArrayList* can hold any number of objects, and it will grow, when necessary, as objects are added to it. Since the items in the list are of type `Object`, the list can actually hold objects of any type.

A program that wants to keep track of various *Shapes* that have been drawn on the screen can store those shapes in an *ArrayList*. Suppose that the *ArrayList* is named `listOfShapes`. A shape, such as `oneShape`, can be added to the end of the list by calling the instance method "`listOfShapes.add(oneShape);`". The shape can be removed from the list with the instance method "`listOfShapes.remove(oneShape);`". The number of shapes in the list is given by the function "`listOfShapes.size()`". And it is possible to retrieve the i-th object from the list with the function call "`listOfShapes.get(i)`". (Items in the list are numbered from 0 to `listOfShapes.size() - 1`.) However, note that this method returns an *Object*, not a *Shape*. (Of course, the people who wrote the `ArrayList` class didn't even know about *Shapes*, so the method they wrote could hardly have a return type of *Shape*!) Since you know that the items in the list are, in fact, *Shapes* and not just *Objects*, you can type-cast the *Object* returned by `listOfShapes.get(i)` to be a value of type *Shape*:

```
oneShape = (Shape)listOfShapes.get(i);
```

Let's say, for example, that you want to redraw all the shapes in the list. You could do this with a simple `for` loop, which is a lovely example of object-oriented programming and of polymorphism:

```
for (int i = 0; i < listOfShapes.size(); i++) {
   Shape s;  // i-th element of the list, considered as a Shape
   s = (Shape)listOfShapes.get(i);
   s.redraw();  // What is drawn here depends on what type of shape s is!
}
```

* * *

The sample source code file *ShapeDraw.java* uses an abstract *Shape* class and an *ArrayList* to hold a list of shapes. The file defines an applet in which the user can add various shapes to a drawing area. Once a shape is in the drawing area, the user can use the mouse to drag it around.

You might want to look at this file, even though you won't be able to understand all of it at this time. Even the definitions of the shape classes are somewhat different from those that I have described in this section. (For example, the `draw()` method has a parameter of type *Graphics*. This parameter is required because of the way Java handles all drawing.) I'll return to similar examples in later chapters when you know more about GUI programming. However, it would still be worthwhile to look at the definition of the *Shape* class and its subclasses in the source code. You might also check how an `ArrayList` is used to hold the list of shapes.

In the applet, the only time when the actual class of a shape is used is when that shape is added to the screen. Once the shape has been created, it is manipulated entirely as an abstract shape. The routine that implements dragging, for example, works with variables of type *Shape* and makes no reference to any of its subclasses. As the shape is being dragged, the dragging routine just calls the shape's draw method each time the shape has to be drawn, so it doesn't have to know how to draw the shape or even what type of shape it is. The object is responsible for drawing itself. If I wanted to add a new type of shape to the program, I would define a new subclass of *Shape*, add another button to the applet, and program the button to add the correct type of shape to the screen. No other changes in the programming would be necessary.

If you want to try out the applet, you can find it at the end of the on-line version of this section.

5.6 this and super

Although the basic ideas of object-oriented programming are reasonably simple and clear, they are subtle, and they take time to get used to. And unfortunately, beyond the basic ideas there are a lot of details. This section and the next cover more of those annoying details. You should not necessarily master everything in these two sections the first time through, but you should read it to be aware of what is possible. For the most part, when I need to use this material later in the text, I will explain it again briefly, or I will refer you back to it. In this section, we'll look at two variables, `this` and `super`, that are automatically defined in any instance method.

5.6.1 The Special Variable this

What does it mean when you use a simple identifier such as `amount` or `process()` to refer to a variable or method? The answer depends on scope rules that tell where and how each declared variable and method can be accessed in a program. Inside the definition of a method, a simple variable name might refer to a local variable or parameter, if there is one "in scope," that is, one whose declaration is in effect at the point in the source code where the reference occurs. If not, it must refer to a member variable of the class in which the reference occurs. Similarly, a simple method name must refer to a method in the same class.

A **static** member of a class has a simple name that can only be used inside the class definition; for use outside the class, it has a full name of the form ⟨*class-name*⟩.⟨*simple-name*⟩. For example, "`Math.PI`" is a static member variable with simple name "`PI`" in the class "*Math*". It's always legal to use the full name of a static member, even within the class where it's defined. Sometimes it's even necessary, as when the simple name of a static member variable is hidden by a local variable or parameter of the same name.

Instance variables and instance methods also have simple names. The simple name of such an instance member can be used in instance methods in the class where the instance member

is defined (but not in static methods). Instance members also have full names—but remember that instance variables and methods are actually contained in objects, not classes. The full name of an instance member starts with a reference to the object that contains the instance member. For example, if `std` is a variable that refers to an object of type *Student*, then `std.test1` could be the full name of an instance variable named `test1` that is contained in that object. Inside the *Student* class, the same variable could be referred to simply as `test1`. But when just the simple name is used, where is the object that contains the variable? As an instance variable, `test1` is not a part of the *Student* class itself; any actual `test1` variable has to be contained in some object of type student.

The solution to this riddle is simple: Suppose that the reference to "`test1`" occurs in the definition of some instance method. As with instance variables, only the definition of the instance method is in the class; the actual method that gets executed has to be thought of as belonging to some particular object of type *Student*. When that method gets executed, the occurrence of the name "`test1`" refers to the `test1` variable **in that same object**. (This is why simple names of instance members cannot be used in static methods—when a static method is executed, there is no object around and hence no actual instance members to refer to!)

This leaves open the question of full names for instance members inside the same class where they are defined. We need a way to refer to "the object that contains this method." Java defines a special variable named ***this*** for just this purpose, which is used in the source code of an instance method to refer to the object that contains the method. This intent of the name, "`this`," is to refer to "this object," the one right here that this very method is in. If `var` is an instance variable in the same object as the method, then "`this.var`" is a full name for that variable. If `otherMethod()` is an instance method in the same object, then `this.otherMethod()` could be used to call that method. Whenever the computer executes an instance method, it automatically sets the variable `this` to refer to the object that contains the method.

One common use of `this` is in constructors. For example:

```
public class Student {

    private String name;  // Name of the student.

    public Student(String name) {
         // Constructor.  Create a student with specified name.
       this.name = name;
    }
      .
      .   // More variables and methods.
      .
}
```

In the constructor, the instance variable called `name` is hidden by a formal parameter. However, the instance variable can still be referred to by its full name, `this.name`. In the assignment statement "`this.name = name`", the value of the formal parameter, `name`, is assigned to the instance variable, `this.name`. This is considered to be acceptable style: There is no need to dream up cute new names for formal parameters that are just used to initialize instance variables. You can use the same name for the parameter as for the instance variable.

There are other uses for `this`. Sometimes, when you are writing an instance method, you need to pass the object that contains the method to a subroutine, as an actual parameter. In that case, you can use `this` as the actual parameter. For example, if you wanted to print out a

string representation of the object, you could say "`System.out.println(this);`". If you want to add it to an *ArrayList* `lst`, you could say `lst.add(this)`. Or you could assign the value of `this` to another variable in an assignment statement. In fact, you can do anything with `this` that you could do with any other variable, except change its value.

5.6.2 The Special Variable super

Java also defines another special variable, named "`super`", for use in the definitions of instance methods. The variable `super` is for use in a subclass. Like `this`, `super` refers to the object that contains the method. But it's forgetful. It forgets that the object belongs to the class you are writing, and it remembers only that it belongs to the superclass of that class. The point is that the class can contain additions and modifications to the superclass. `super` doesn't know about any of those additions and modifications; it can only be used to refer to methods and variables in the superclass.

Let's say that the class that you are writing contains an instance method named `doSomething()`. Consider the subroutine call statement `super.doSomething()`. Now, `super` doesn't know anything about the `doSomething()` method in the subclass. It only knows about things in the superclass, so it tries to execute a method named `doSomething()` from the superclass. If there is none—if the `doSomething()` method was an addition rather than a modification—you'll get a syntax error.

The reason `super` exists is so you can get access to things in the superclass that are **hidden** by things in the subclass. For example, `super.var` always refers to an instance variable named `var` in the superclass. This can be useful for the following reason: If a class contains an instance variable with the same name as an instance variable in its superclass, then an object of that class will actually contain two variables with the same name: one defined as part of the class itself and one defined as part of the superclass. The variable in the subclass does not **replace** the variable of the same name in the superclass; it merely ***hides*** it. The variable from the superclass can still be accessed, using `super`.

When you write a method in a subclass that has the same signature as a method in its superclass, the method from the superclass is hidden in the same way. We say that the method in the subclass ***overrides*** the method from the superclass. Again, however, `super` can be used to access the method from the superclass.

The major use of `super` is to override a method with a new method that **extends** the behavior of the inherited method, instead of **replacing** that behavior entirely. The new method can use `super` to call the method from the superclass, and then it can add additional code to provide additional behavior. As an example, suppose you have a *PairOfDice* class that includes a `roll()` method. Suppose that you want a subclass, *GraphicalDice*, to represent a pair of dice drawn on the computer screen. The `roll()` method in the *GraphicalDice* class should do everything that the `roll()` method in the *PairOfDice* class does. We can express this with a call to `super.roll()`, which calls the method in the superclass. But in addition to that, the `roll()` method for a *GraphicalDice* object has to redraw the dice to show the new values. The *GraphicalDice* class might look something like this:

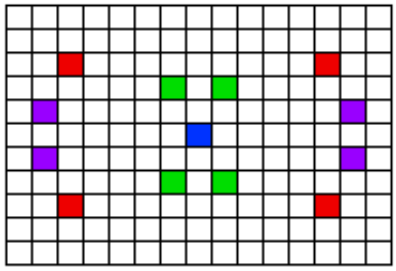
```
public class GraphicalDice extends PairOfDice {

    public void roll() {
            // Roll the dice, and redraw them.
         super.roll();  // Call the roll method from PairOfDice.
         redraw();      // Call a method to draw the dice.
```

```
        }
            .
            .  // More stuff, including definition of redraw().
            .
    }
```

Note that this allows you to extend the behavior of the `roll()` method even if you don't know how the method is implemented in the superclass!

Here is a more complete example. The applet at the end of Section 4.7 in the on-line version of this book shows a disturbance that moves around in a mosaic of little squares. As it moves, each square that it visits becomes a brighter shade of green. The result looks interesting, but I think it would be prettier if the pattern were symmetric. A symmetric version of the applet is shown at the bottom of Section 5.7 (the on-line version). The symmetric applet can be programmed as an easy extension of the original applet.

In the symmetric version, each time a square is brightened, the squares that can be obtained from that one by horizontal and vertical reflection through the center of the mosaic are also brightened. This picture might make the symmetry idea clearer:



The four red squares in the picture, for example, form a set of such symmetrically placed squares, as do the purple squares and the green squares. (The blue square is at the center of the mosaic, so reflecting it doesn't produce any other squares; it's its own reflection.)

The original applet is defined by the class *RandomBrighten*. In that class, the actual task of brightening a square is done by a method called `brighten()`. If `row` and `col` are the row and column numbers of a square, then "`brighten(row,col);`" increases the brightness of that square. All we need is a subclass of *RandomBrighten* with a modified `brighten()` routine. Instead of just brightening one square, the modified routine will also brighten the horizontal and vertical reflections of that square. But how will it brighten each of the four individual squares? By calling the `brighten()` method from the original class! It can do this by calling `super.brighten()`.

There is still the problem of computing the row and column numbers of the horizontal and vertical reflections. To do this, you need to know the number of rows and the number of columns. The *RandomBrighten* class has instance variables named `ROWS` and `COLUMNS` to represent these quantities. Using these variables, it's possible to come up with formulas for the reflections, as shown in the definition of the `brighten()` method below.

Here's the complete definition of the new class:

```
public class SymmetricBrighten extends RandomBrighten {

   /**
    * Brighten the specified square, at position (row,col) and its
    * horizontal and vertical reflections.  This overrides the
```

```
     * brighten() method from the RandomBrighten class, which just
     * brightens one square.
     */
    void brighten(int row, int col) {
       super.brighten(row, col);
       super.brighten(ROWS - 1 - row, col);
       super.brighten(row, COLUMNS - 1 - col);
       super.brighten(ROWS - 1 - row, COLUMNS - 1 - col);
    }

} // end class SymmetricBrighten
```

This is the entire source code for the applet!

5.6.3 Constructors in Subclasses

Constructors are not inherited. That is, if you extend an existing class to make a subclass, the constructors in the superclass do `not` become part of the subclass. If you want constructors in the subclass, you have to define new ones from scratch. If you don't define any constructors in the subclass, then the computer will make up a default constructor, with no parameters, for you.

This could be a problem, if there is a constructor in the superclass that does a lot of necessary work. It looks like you might have to repeat all that work in the subclass! This could be a **real** problem if you don't have the source code to the superclass, and don't know how it works. It might look like an impossible problem, if the constructor in the superclass uses `private` member variables that you don't even have access to in the subclass!

Obviously, there has to be some fix for this, and there is. It involves the special variable, `super`. As the very first statement in a constructor, you can use `super` to call a constructor from the superclass. The notation for this is a bit ugly and misleading, and it can only be used in this one particular circumstance: It looks like you are calling `super` as a subroutine (even though `super` is not a subroutine and you can't call constructors the same way you call other subroutines anyway). As an example, assume that the *PairOfDice* class has a constructor that takes two integers as parameters. Consider a subclass:

```
public class GraphicalDice extends PairOfDice {

     public GraphicalDice() {  // Constructor for this class.

         super(3,4);  // Call the constructor from the
                      //   PairOfDice class, with parameters 3, 4.

         initializeGraphics();  // Do some initialization specific
                                //   to the GraphicalDice class.
     }
        .
        .  // More constructors, methods, variables...
        .
}
```

The statement "`super(3,4);`" calls the constructor from the superclass. This call must be the first line of the constructor in the subclass. Note that if you don't explicitly call a constructor from the superclass in this way, then the default constructor from the superclass, the one with no parameters, will be called automatically. (And if no such constructor exists in the superclass, the compiler will consider it to be a syntax error.)

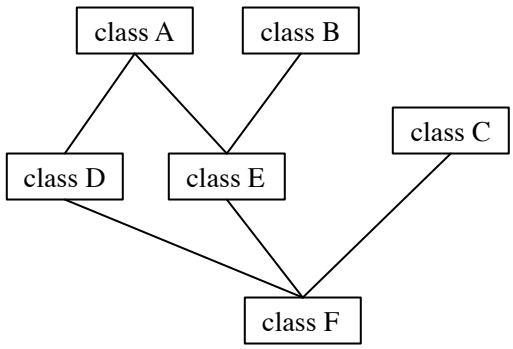
This might seem rather technical, but unfortunately it is sometimes necessary. By the way, you can use the special variable `this` in exactly the same way to call another constructor in the same class. This can be useful since it can save you from repeating the same code in several different constructors.

5.7 Interfaces, Nested Classes, and Other Details

THIS SECTION simply pulls together a few more miscellaneous features of object oriented programming in Java. Read it now, or just look through it and refer back to it later when you need this material. (You will need to know about the first topic, interfaces, almost as soon as we begin GUI programming.)

5.7.1 Interfaces

Some object-oriented programming languages, such as C++, allow a class to extend two or more superclasses. This is called ***multiple inheritance***. In the illustration below, for example, class E is shown as having both class A and class B as direct superclasses, while class F has three direct superclasses.



Multiple inheritance (NOT allowed in Java)

Such multiple inheritance is **not** allowed in Java. The designers of Java wanted to keep the language reasonably simple, and felt that the benefits of multiple inheritance were not worth the cost in increased complexity. However, Java does have a feature that can be used to accomplish many of the same goals as multiple inheritance: ***interfaces***.

We've encountered the term "interface" before, in connection with black boxes in general and subroutines in particular. The interface of a subroutine consists of the name of the subroutine, its return type, and the number and types of its parameters. This is the information you need to know if you want to call the subroutine. A subroutine also has an implementation: the block of code which defines it and which is executed when the subroutine is called.

In Java, `interface` is a reserved word with an additional, technical meaning. An "`interface`" in this sense consists of a set of instance method interfaces, without any associated implementations. (Actually, a Java interface can contain other things as well, but we won't discuss them here.) A class can ***implement*** an `interface` by providing an implementation for each of the methods specified by the interface. Here is an example of a very simple Java `interface`:

```
public interface Drawable {
   public void draw(Graphics g);
}
```

This looks much like a class definition, except that the implementation of the `draw()` method is omitted. A class that implements the `interface` *Drawable* must provide an implementation for this method. Of course, the class can also include other methods and variables. For example,

```
public class Line implements Drawable {
    public void draw(Graphics g) {
        . . . // do something---presumably, draw a line
    }
    . . . // other methods and variables
}
```

Note that to implement an interface, a class must do more than simply provide an implementation for each method in the interface; it must also **state** that it implements the interface, using the reserved word `implements` as in this example: "`public class Line implements Drawable`". Any class that implements the *Drawable* interface defines a `draw()` instance method. Any object created from such a class includes a `draw()` method. We say that an **object** implements an `interface` if it belongs to a class that implements the interface. For example, any object of type *Line* implements the *Drawable* interface.

While a class can **extend** only one other class, it can **implement** any number of interfaces. In fact, a class can both extend one other class and implement one or more interfaces. So, we can have things like

```
class FilledCircle extends Circle
                        implements Drawable, Fillable {
   . . .
}
```

The point of all this is that, although interfaces are not classes, they are something very similar. An interface is very much like an abstract class, that is, a class that can never be used for constructing objects, but can be used as a basis for making subclasses. The subroutines in an interface are abstract methods, which must be implemented in any concrete class that implements the interface. You can compare the *Drawable* interface with the abstract class

```
public abstract class AbstractDrawable {
   public abstract void draw(Graphics g);
}
```

The main difference is that a class that extends *AbstractDrawable* cannot extend any other class, while a class that implements *Drawable* can also extend some class. As with abstract classes, even though you can't construct an object from an interface, you can declare a variable whose type is given by the interface. For example, if *Drawable* is an interface, and if *Line* and *FilledCircle* are classes that implement *Drawable*, then you could say:

```
Drawable figure;  // Declare a variable of type Drawable.  It can
                  //    refer to any object that implements the
                  //    Drawable interface.

figure = new Line();  // figure now refers to an object of class Line
figure.draw(g);   // calls draw() method from class Line

figure = new FilledCircle();   // Now, figure refers to an object
                               //   of class FilledCircle.
figure.draw(g);   // calls draw() method from class FilledCircle
```

A variable of type *Drawable* can refer to any object of any class that implements the *Drawable* interface. A statement like `figure.draw(g)`, above, is legal because `figure` is of type *Drawable*, and **any** *Drawable* object has a `draw()` method. So, whatever object `figure` refers to, that object must have a `draw()` method.

Note that a ***type*** is something that can be used to declare variables. A type can also be used to specify the type of a parameter in a subroutine, or the return type of a function. In Java, a type can be either a class, an interface, or one of the eight built-in primitive types. These are the only possibilities. Of these, however, only classes can be used to construct new objects.

You are not likely to need to write your own interfaces until you get to the point of writing fairly complex programs. However, there are several interfaces that are used in important ways in Java's standard packages. You'll learn about some of these standard interfaces in the next few chapters, and you will write classes that implement them.

5.7.2 Nested Classes

A class seems like it should be a pretty important thing. A class is a high-level building block of a program, representing a potentially complex idea and its associated data and behaviors. I've always felt a bit silly writing tiny little classes that exist only to group a few scraps of data together. However, such trivial classes are often useful and even essential. Fortunately, in Java, I can ease the embarrassment, because one class can be nested inside another class. My trivial little class doesn't have to stand on its own. It becomes part of a larger more respectable class. This is particularly useful when you want to create a little class specifically to support the work of a larger class. And, more seriously, there are other good reasons for nesting the definition of one class inside another class.

In Java, a ***nested class*** is any class whose definition is inside the definition of another class. Nested classes can be either ***named*** or ***anonymous***. I will come back to the topic of anonymous classes later in this section. A named nested class, like most other things that occur in classes, can be either static or non-static.

The definition of a static nested class looks just like the definition of any other class, except that it is nested inside another class and it has the modifier `static` as part of its declaration. A static nested class is part of the static structure of the containing class. It can be used inside that class to create objects in the usual way. If it has not been declared private, then it can also be used outside the containing class, but when it is used outside the class, its name must indicate its membership in the containing class. This is similar to other static components of a class: A static nested class is part of the class itself in the same way that static member variables are parts of the class itself.

For example, suppose a class named *WireFrameModel* represents a set of lines in three-dimensional space. (Such models are used to represent three-dimensional objects in graphics programs.) Suppose that the *WireFrameModel* class contains a static nested class, *Line*, that represents a single line. Then, outside of the class *WireFrameModel*, the *Line* class would be referred to as `WireFrameModel.Line`. Of course, this just follows the normal naming convention for static members of a class. The definition of the *WireFrameModel* class with its nested *Line* class would look, in outline, like this:

```
public class WireFrameModel {

   . . . // other members of the WireFrameModel class

   static public class Line {
```

```
            // Represents a line from the point (x1,y1,z1)
            // to the point (x2,y2,z2) in 3-dimensional space.
         double x1, y1, z1;
         double x2, y2, z2;
      } // end class Line

      . . . // other members of the WireFrameModel class

   } // end WireFrameModel
```

Inside the *WireFrameModel* class, a *Line* object would be created with the constructor "`new Line()`". Outside the class, "`new WireFrameModel.Line()`" would be used.

A static nested class has full access to the static members of the containing class, even to the private members. Similarly, the containing class has full access to the members of the nested class. This can be another motivation for declaring a nested class, since it lets you give one class access to the private members of another class without making those members generally available to other classes. Note also that a nested class can itself be private, meaning that it can only be used inside the class in which it is nested.

When you compile the above class definition, two class files will be created. Even though the definition of *Line* is nested inside *WireFrameModel*, the compiled *Line* class is stored in a separate file. The name of the class file for *Line* will be `WireFrameModel$Line.class`.

* * *

Non-static nested classes are referred to as ***inner classes***. Inner classes are not, in practice, very different from static nested classes, but a non-static nested class is actually associated with an object rather than to the class in which it is nested. This can take some getting used to.

Any non-static member of a class is not really part of the class itself (although its source code is contained in the class definition). This is true for inner classes, just as it is for any other non-static part of a class. The non-static members of a class specify what will be contained in objects that are created from that class. The same is true—at least logically—for inner classes. It's as if each object that belongs to the containing class has its **own copy** of the nested class. This copy has access to all the instance methods and instance variables of the object, even to those that are declared `private`. The two copies of the inner class in two different objects differ because the instance variables and methods they refer to are in different objects. In fact, the rule for deciding whether a nested class should be static or non-static is simple: If the nested class needs to use any instance variable or instance method from the containing class, make the nested class non-static. Otherwise, it might as well be static.

From outside the containing class, a non-static nested class has to be referred to using a name of the form ⟨*variableName*⟩.⟨*NestedClassName*⟩, where ⟨*variableName*⟩ is a variable that refers to the object that contains the class. This is actually rather rare, however. A non-static nested class is generally used only inside the class in which it is nested, and there it can be referred to by its simple name.

In order to create an object that belongs to an inner class, you must first have an object that belongs to the containing class. (When working inside the class, the object "`this`" is used implicitly.) The inner class object is permanently associated with the containing class object, and it has complete access to the members of the containing class object. Looking at an example will help, and will hopefully convince you that inner classes are really very natural. Consider a class that represents poker games. This class might include a nested class to represent the players of the game. This structure of the *PokerGame* class could be:

```
public class PokerGame {  // Represents a game of poker.
```

```
        class Player {  // Represents one of the players in this game.
           .
           .
           .
        } // end class Player

        private Deck deck;      // A deck of cards for playing the game.
        private int pot;        // The amount of money that has been bet.

        .
        .
        .
    } // end class PokerGame
```

If `game` is a variable of type *PokerGame*, then, conceptually, `game` contains its own copy of the *Player* class. In an instance method of a *PokerGame* object, a new *Player* object would be created by saying "`new Player()`", just as for any other class. (A *Player* object could be created outside the *PokerGame* class with an expression such as "`game.new Player()`". Again, however, this is rare.) The *Player* object will have access to the `deck` and `pot` instance variables in the *PokerGame* object. Each `PokerGame` object has its own `deck` and `pot` and `Players`. Players of that poker game use the deck and pot for that game; players of another poker game use the other game's deck and pot. That's the effect of making the *Player* class non-static. This is the most natural way for players to behave. A *Player* object represents a player of one particular poker game. If *Player* were a **static** nested class, on the other hand, it would represent the general idea of a poker player, independent of a particular poker game.

5.7.3 Anonymous Inner Classes

In some cases, you might find yourself writing an inner class and then using that class in just a single line of your program. Is it worth creating such a class? Indeed, it can be, but for cases like this you have the option of using an ***anonymous inner class***. An anonymous class is created with a variation of the `new` operator that has the form

```
new ⟨superclass-or-interface⟩ ( ⟨parameter-list⟩ ) {
        ⟨methods-and-variables⟩
    }
```

This constructor defines a new class, without giving it a name, and it simultaneously creates an object that belongs to that class. This form of the `new` operator can be used in any statement where a regular "`new`" could be used. The intention of this expression is to create: "a new object belonging to a class that is the same as ⟨*superclass-or-interface*⟩ but with these ⟨*methods-and-variables*⟩ added." The effect is to create a uniquely customized object, just at the point in the program where you need it. Note that it is possible to base an anonymous class on an interface, rather than a class. In this case, the anonymous class must implement the interface by defining all the methods that are declared in the interface. If an interface is used as a base, the ⟨*parameter-list*⟩ must be empty. Otherwise, it can contain parameters for a constructor in the ⟨*superclass*⟩.

Anonymous classes are often used for handling events in graphical user interfaces, and we will encounter them several times in the chapters on GUI programming. For now, we will look at one not-very-plausible example. Consider the *Drawable* interface, which is defined earlier in

this section. Suppose that we want a *Drawable* object that draws a filled, red, 100-pixel square. Rather than defining a new, separate class and then using that class to create the object, we can use an anonymous class to create the object in one statement:

```
Drawable redSquare = new Drawable() {
       void draw(Graphics g) {
          g.setColor(Color.red);
          g.fillRect(10,10,100,100);
       }
   };
```

The semicolon at the end of this statement is not part of the class definition. It's the semicolon that is required at the end of every declaration statement.

When a Java class is compiled, each anonymous nested class will produce a separate class file. If the name of the main class is `MainClass`, for example, then the names of the class files for the anonymous nested classes will be `MainClass$1.class`, `MainClass$2.class`, `MainClass$3.class`, and so on.

5.7.4 Mixing Static and Non-static

Classes, as I've said, have two very distinct purposes. A class can be used to group together a set of static member variables and static methods. Or it can be used as a factory for making objects. The non-static variables and methods in the class definition specify the instance variables and methods of the objects. In most cases, a class performs one or the other of these roles, not both.

Sometimes, however, static and non-static members are mixed in a single class. In this case, the class plays a dual role. Sometimes, these roles are completely separate. But it is also possible for the static and non-static parts of a class to interact. This happens when instance methods use static member variables or call static member subroutines. An instance method belongs to an object, not to the class itself, and there can be many objects with their own versions of the instance method. But there is only one copy of a static member variable. So, effectively, we have many objects sharing that one variable.

Suppose, for example, that we want to write a *PairOfDice* class that uses the *Random* class mentioned in Section 5.3 for rolling the dice. To do this, a *PairOfDice* object needs access to an object of type *Random*. But there is no need for each *PairOfDice* object to have a separate *Random* object. (In fact, it would not even be a good idea: Because of the way random number generators work, a program should, in general, use only one source of random numbers.) A nice solution is to have a single *Random* variable as a `static` member of the *PairOfDice* class, so that it can be shared by all *PairOfDice* objects. For example:

```
import java.util.Random;

public class PairOfDice {

    private static Random randGen = new Random();

    public int die1;   // Number showing on the first die.
    public int die2;   // Number showing on the second die.

    public PairOfDice() {
            // Constructor.  Creates a pair of dice that
            // initially shows random values.
         roll();
```

```
    }

    public void roll() {
            // Roll the dice by setting each of the dice to be
            // a random number between 1 and 6.
         die1 = randGen.nextInt(6) + 1;  // Use the static variable!
         die2 = randGen.nextInt(6) + 1;
    }

} // end class PairOfDice
```

As another example, let's rewrite the *Student* class that was used in Section 5.2. I've added an `ID` for each student and a `static` member called `nextUniqueID`. Although there is an `ID` variable in each student object, there is only one `nextUniqueID` variable.

```
public class Student {

   private String name;  // Student's name.
   private int ID;  // Unique ID number for this student.
   public double test1, test2, test3;   // Grades on three tests.

   private static int nextUniqueID = 0;
             // keep track of next available unique ID number

   Student(String theName) {
        // Constructor for Student objects; provides a name for the Student,
        // and assigns the student a unique ID number.
      name = theName;
      nextUniqueID++;
      ID = nextUniqueID;
   }

   public String getName() {
        // Accessor method for reading the value of the private
        // instance variable, name.
      return name;
   }

   public int getID() {
        // Accessor method for reading the value of ID.
      return ID;
   }

   public double getAverage() {
        // Compute average test grade.
      return (test1 + test2 + test3) / 3;
   }

}  // end of class Student
```

Since `nextUniqueID` is a `static` variable, the initialization "`nextUniqueID = 0`" is done only once, when the class is first loaded. Whenever a *Student* object is constructed and the constructor says "`nextUniqueID++;`", it's always the same static member variable that is being incremented. When the very first *Student* object is created, `nextUniqueID` becomes 1. When the second object is created, `nextUniqueID` becomes 2. After the third object, it becomes 3. And so on. The constructor stores the new value of `nextUniqueID` in the `ID` variable of the object that is being created. Of course, `ID` is an instance variable, so every object has its own

individual `ID` variable. The class is constructed so that each student will automatically get a different value for its `ID` variable. Furthermore, the `ID` variable is `private`, so there is no way for this variable to be tampered with after the object has been created. You are guaranteed, just by the way the class is designed, that every student object will have its own permanent, unique identification number. Which is kind of cool if you think about it.

(Unfortunately, if you think about it a bit more, it turns out that the guarantee isn't quite absolute. The guarantee is valid in programs that use a single thread. But, as a preview of the difficulties of parallel programming, I'll note that in multi-threaded programs, where several things can be going on at the same time, things can get a bit strange. In a multi-threaded program, it is possible that two threads are creating *Student* objects at exactly the same time, and it becomes possible for both objects to get the same ID number. We'll come back to this in Subsection 12.1.3, where you will learn how to fix the problem.)

5.7.5 Static Import

The `import` directive makes it possible to refer to a class such as `java.awt.Color` using its simple name, *Color*. All you have to do is say `import java.awt.Color` or `import java.awt.*`. But you still have to use compound names to refer to static member variables such as `System.out` and to static methods such as `Math.sqrt`.

Java 5.0 introduced a new form of the `import` directive that can be used to import `static` members of a class in the same way that the ordinary `import` directive imports classes from a package. The new form of the directive is called a ***static import***, and it has syntax

```
import static ⟨package-name⟩.⟨class-name⟩.⟨static-member-name⟩;
```

to import one static member name from a class, or

```
import static ⟨package-name⟩.⟨class-name⟩.*;
```

to import all the public static members from a class. For example, if you preface a class definition with

```
import static java.lang.System.out;
```

then you can use the simple name `out` instead of the compound name `System.out`. This means you can use `out.println` instead of `System.out.println`. If you are going to work extensively with the *Math* class, you can preface your class definition with

```
import static java.lang.Math.*;
```

This would allow you to say `sqrt` instead of `Math.sqrt`, `log` instead of `Math.log`, `PI` instead of `Math.PI`, and so on.

Note that the static import directive requires a ⟨*package-name*⟩, even for classes in the standard package `java.lang`. One consequence of this is that you can't do a static import from a class in the default package. In particular, it is not possible to do a static import from my *TextIO* class—if you wanted to do that, you would have to move *TextIO* into a package.

5.7.6 Enums as Classes

Enumerated types were introduced in Subsection 2.3.3. Now that we have covered more material on classes and objects, we can revisit the topic (although still not covering enumerated types in their full complexity).

Enumerated types are actually classes, and each enumerated type constant is a `public`, `final`, `static` member variable in that class (even though they are not declared with these modifiers). The value of the variable is an object belonging to the enumerated type class. There is one such object for each enumerated type constant, and these are the only objects of the class that can ever be created. It is really these objects that represent the possible values of the enumerated type. The enumerated type constants are actually variables that refer to these objects.

When an enumerated type is defined inside another class, it is a nested class inside the enclosing class. In fact, it is a static nested class, whether you declare it to be `static` or not. But it can also be declared as a non-nested class, in a file of its own. For example, we could define the following enumerated type in a file named `Suit.java`:

```
public enum Suit {

   SPADES, HEARTS, DIAMONDS, CLUBS

}
```

This enumerated type represents the four possible suits for a playing card, and it could have been used in the example *Card.java* from Subsection 5.4.2.

Furthermore, in addition to its list of values, an enumerated type can contain some of the other things that a regular class can contain, including methods and additional member variables. Just add a semicolon (;) at the end of the list of values, and then add definitions of the methods and variables in the usual way. For example, we might make an enumerated type to represent the possible values of a playing card. It might be useful to have a method that returns the corresponding value in the game of Blackjack. As another example, suppose that when we print out one of the values, we'd like to see something different from the default string representation (the identifier that names the constant). In that case, we can override the `toString()` method in the class to print out a different string representation. This would give something like:

```
public enum CardValue {

   ACE, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT,
         NINE, TEN, JACK, QUEEN, KING;

   /**
    * Return the value of this CardValue in the game of Blackjack.
    * Note that the value returned for an ace is 1.
    */
   public int blackJackValue() {
      if (this == JACK || this == QUEEN || this == KING)
         return 10;
      else
         return 1 + ordinal();
   }

   /**
    * Return a String representation of this CardValue, using numbers
    * for the numerical cards and names for the ace and face cards.
    */
   public String toString() {
      switch (this) {       // "this" is one of the enumerated type values
      case ACE:
```

```
            return "Ace";
         case JACK:
            return "Jack";
         case QUEEN:
            return "Queen";
         case KING:
            return "King";
         default:              // it's a numeric card value
            int numericValue = 1 + ordinal();
            return "" + numericValue;
      }

   } // end CardValue
```

The methods `blackjackValue()` and `toString()` are instance methods in *CardValue*. Since `CardValue.JACK` is an object belonging to that class, you can call `CardValue.JACK.blackjackValue()`. Suppose that `cardVal` is declared to be a variable of type *CardValue*, so that it can refer to any of the values in the enumerated type. We can call `cardVal.blackjackValue()` to find the Blackjack value of the *CardValue* object to which `cardVal` refers, and `System.out.println(cardVal)` will implicitly call the method `cardVal.toString()` to obtain the print representation of that *CardValue*. (One other thing to keep in mind is that since *CardValue* is a class, the value of `cardVal` can be `null`, which means it does not refer to any object.)

Remember that `ACE`, `TWO`, ..., `KING` are the only possible objects of type *CardValue*, so in an instance method in that class, `this` will refer to one of those values. Recall that the instance method `ordinal()` is defined in any enumerated type and gives the position of the enumerated type value in the list of possible values, with the count starting from zero.

(If you find it annoying to use the class name as part of the name of every enumerated type constant, you can use static import to make the simple names of the constants directly available—but only if you put the enumerated type into a package. For example, if the enumerated type *CardValue* is defined in a package named `cardgames`, then you could place

```
import static cardgames.CardValue.*;
```

at the beginning of a source code file. This would allow you, for example, to use the name `JACK` in that file instead of `CardValue.JACK`.)

Exercises for Chapter 5

1. In all versions of the *PairOfDice* class in Section 5.2, the instance variables `die1` and `die2` are declared to be `public`. They really should be private, so that they would be protected from being changed from outside the class. Write another version of the *PairOfDice* class in which the instance variables `die1` and `die2` are `private`. Your class will need "getter" methods that can be used to find out the values of `die1` and `die2`. (The idea is to protect their values from being changed from outside the class, but still to allow the values to be read.) Include other improvements in the class, if you can think of any. Test your class with a short program that counts how many times a pair of dice is rolled, before the total of the two dice is equal to two.

2. A common programming task is computing statistics of a set of numbers. (A statistic is a number that summarizes some property of a set of data.) Common statistics include the mean (also known as the average) and the standard deviation (which tells how spread out the data are from the mean). I have written a little class called *StatCalc* that can be used to compute these statistics, as well as the sum of the items in the dataset and the number of items in the dataset. You can read the source code for this class in the file *StatCalc.java*. If `calc` is a variable of type *StatCalc*, then the following methods are defined:

- `calc.enter(item)` where `item` is a number, adds the item to the dataset.
- `calc.getCount()` is a function that returns the number of items that have been added to the dataset.
- `calc.getSum()` is a function that returns the sum of all the items that have been added to the dataset.
- `calc.getMean()` is a function that returns the average of all the items.
- `calc.getStandardDeviation()` is a function that returns the standard deviation of the items.

Typically, all the data are added one after the other by calling the `enter()` method over and over, as the data become available. After all the data have been entered, any of the other methods can be called to get statistical information about the data. The methods `getMean()` and `getStandardDeviation()` should only be called if the number of items is greater than zero.

Modify the current source code, *StatCalc.java*, to add instance methods `getMax()` and `getMin()`. The `getMax()` method should return the largest of all the items that have been added to the dataset, and `getMin()` should return the smallest. You will need to add two new instance variables to keep track of the largest and smallest items that have been seen so far.

Test your new class by using it in a program to compute statistics for a set of non-zero numbers entered by the user. Start by creating an object of type *StatCalc*:

```
StatCalc  calc;   // Object to be used to process the data.
calc = new StatCalc();
```

Read numbers from the user and add them to the dataset. Use 0 as a sentinel value (that is, stop reading numbers when the user enters 0). After all the user's non-zero

numbers have been entered, print out each of the six statistics that are available from `calc`.

3. This problem uses the *PairOfDice* class from Exercise 5.1 and the *StatCalc* class from Exercise 5.2.

The program in Exercise 4.4 performs the experiment of counting how many times a pair of dice is rolled before a given total comes up. It repeats this experiment 10000 times and then reports the average number of rolls. It does this whole process for each possible total (2, 3, . . . , 12).

Redo that exercise. But instead of just reporting the average number of rolls, you should also report the standard deviation and the maximum number of rolls. Use a *PairOfDice* object to represent the dice. Use a *StatCalc* object to compute the statistics. (You'll need a new *StatCalc* object for each possible total, 2, 3, . . . , 12. You can use a new pair of dice if you want, but it's not necessary.)

4. The *BlackjackHand* class from Subsection 5.5.1 is an extension of the *Hand* class from Section 5.4. The instance methods in the *Hand* class are discussed in that section. In addition to those methods, *BlackjackHand* includes an instance method, `getBlackjackValue()`, that returns the value of the hand for the game of Blackjack. For this exercise, you will also need the *Deck* and *Card* classes from Section 5.4.

A Blackjack hand typically contains from two to six cards. Write a program to test the *BlackjackHand* class. You should create a *BlackjackHand* object and a *Deck* object. Pick a random number between 2 and 6. Deal that many cards from the deck and add them to the hand. Print out all the cards in the hand, and then print out the value computed for the hand by `getBlackjackValue()`. Repeat this as long as the user wants to continue.

In addition to *TextIO.java*, your program will depend on *Card.java*, *Deck.java*, *Hand.java*, and *BlackjackHand.java*.

5. Write a program that lets the user play Blackjack. The game will be a simplified version of Blackjack as it is played in a casino. The computer will act as the dealer. As in the previous exercise, your program will need the classes defined in *Card.java*, *Deck.java*, *Hand.java*, and *BlackjackHand.java*. (This is the longest and most complex program that has come up so far in the exercises.)

You should first write a subroutine in which the user plays one game. The subroutine should return a **boolean** value to indicate whether the user wins the game or not. Return `true` if the user wins, `false` if the dealer wins. The program needs an object of class *Deck* and two objects of type *BlackjackHand*, one for the dealer and one for the user. The general object in Blackjack is to get a hand of cards whose value is as close to 21 as possible, without going over. The game goes like this.

- First, two cards are dealt into each player's hand. If the dealer's hand has a value of 21 at this point, then the dealer wins. Otherwise, if the user has 21, then the user wins. (This is called a "Blackjack".) Note that the dealer wins on a tie, so if both players have Blackjack, then the dealer wins.
- Now, if the game has not ended, the user gets a chance to add some cards to her hand. In this phase, the user sees her own cards and sees **one** of the dealer's two cards. (In a casino, the dealer deals himself one card face up and one card face down. All the user's cards are dealt face up.) The user makes a decision whether to "Hit",

which means to add another card to her hand, or to "Stand", which means to stop taking cards.

- If the user Hits, there is a possibility that the user will go over 21. In that case, the game is over and the user loses. If not, then the process continues. The user gets to decide again whether to Hit or Stand.
- If the user Stands, the game will end, but first the dealer gets a chance to draw cards. The dealer only follows rules, without any choice. The rule is that as long as the value of the dealer's hand is less than or equal to 16, the dealer Hits (that is, takes another card). The user should see all the dealer's cards at this point. Now, the winner can be determined: If the dealer has gone over 21, the user wins. Otherwise, if the dealer's total is greater than or equal to the user's total, then the dealer wins. Otherwise, the user wins.

Two notes on programming: At any point in the subroutine, as soon as you know who the winner is, you can say "`return true;`" or "`return false;`" to end the subroutine and return to the main program. To avoid having an overabundance of variables in your subroutine, remember that a function call such as `userHand.getBlackjackValue()` can be used anywhere that a number could be used, including in an output statement or in the condition of an `if` statement.

Write a main program that lets the user play several games of Blackjack. To make things interesting, give the user 100 dollars, and let the user make bets on the game. If the user loses, subtract the bet from the user's money. If the user wins, add an amount equal to the bet to the user's money. End the program when the user wants to quit or when she runs out of money.

An applet version of this program can be found in the on-line version of this exercise. You might want to try it out before you work on the program.

6. Subsection 5.7.6 discusses the possibility of representing the suits and values of playing cards as enumerated types. Rewrite the *Card* class from Subsection 5.4.2 to use these enumerated types. Test your class with a program that prints out the 52 possible playing cards. Suggestions: You can modify the source code file *Card.java*, but you should leave out support for Jokers. In your main program, use nested `for` loops to generated cards of all possible suits and values; the `for` loops will be "for-each" loops of the type discussed in Subsection 3.4.4. It would be nice to add a `toString()` method to the *Suit* class from Subsection 5.7.6, so that a suit prints out as "Spades" or "Hearts" instead of "SPADES" or "HEARTS".

Quiz on Chapter 5

1. Object-oriented programming uses *classes* and *objects*. What are classes and what are objects? What is the relationship between classes and objects?

2. Explain carefully what *null* means in Java, and why this special value is necessary.

3. What is a *constructor?* What is the purpose of a constructor in a class?

4. Suppose that `Kumquat` is the name of a class and that `fruit` is a variable of type `Kumquat`. What is the meaning of the statement "`fruit = new Kumquat();`"? That is, what does the computer do when it executes this statement? (Try to give a complete answer. The computer does several things.)

5. What is meant by the terms *instance variable* and *instance method*?

6. Explain what is meant by the terms *subclass* and *superclass.*

7. Modify the following class so that the two instance variables are `private` and there is a getter method and a setter method for each instance variable:

```
public class Player {
   String name;
   int score;
}
```

8. Explain why the class *Player* that is defined in the previous question has an instance method named `toString()`, even though no definition of this method appears in the definition of the class.

9. Explain the term *polymorphism.*

10. Java uses "garbage collection" for memory management. Explain what is meant here by garbage collection. What is the alternative to garbage collection?

11. For this problem, you should write a very simple but complete class. The class represents a counter that counts 0, 1, 2, 3, 4, ... The name of the class should be `Counter`. It has one `private` instance variable representing the value of the counter. It has two instance methods: `increment()` adds one to the counter value, and `getValue()` returns the current counter value. Write a complete definition for the class, `Counter`.

12. This problem uses the `Counter` class from the previous question. The following program segment is meant to simulate tossing a coin 100 times. It should use two `Counter` objects, `headCount` and `tailCount`, to count the number of heads and the number of tails. Fill in the blanks so that it will do so:

```
Counter headCount, tailCount;
tailCount = new Counter();
headCount = new Counter();
for ( int flip = 0;  flip < 100;  flip++ ) {
   if (Math.random() < 0.5)    // There's a 50/50 chance that this is true.

       ____________________ ;   // Count a "head".

   else

       ____________________ ;   // Count a "tail".
}

System.out.println("There were " + ___________________ + " heads.");

System.out.println("There were " + ___________________ + " tails.");
```

Chapter 6

Introduction to GUI Programming

COMPUTER USERS TODAY EXPECT to interact with their computers using a graphical user interface (GUI). Java can be used to write GUI programs ranging from simple applets which run on a Web page to sophisticated stand-alone applications.

GUI programs differ from traditional "straight-through" programs that you have encountered in the first few chapters of this book. One big difference is that GUI programs are ***event-driven***. That is, user actions such as clicking on a button or pressing a key on the keyboard generate events, and the program must respond to these events as they occur. Event-driven programming builds on all the skills you have learned in the first five chapters of this text. You need to be able to write the methods that respond to events. Inside those methods, you are doing the kind of programming-in-the-small that was covered in Chapter 2 and Chapter 3.

And of course, objects are everywhere in GUI programming. Events are objects. Colors and fonts are objects. GUI components such as buttons and menus are objects. Events are handled by instance methods contained in objects. In Java, GUI programming is object-oriented programming.

This chapter covers the basics of GUI programming. The discussion will continue in Chapter 13 with more details and with more advanced techniques.

6.1 The Basic GUI Application

THERE ARE TWO BASIC TYPES of GUI program in Java: ***stand-alone applications*** and ***applets***. An applet is a program that runs in a rectangular area on a Web page. Applets are generally small programs, meant to do fairly simple things, although there is nothing to stop them from being very complex. Applets were responsible for a lot of the initial excitement about Java when it was introduced, since they could do things that could not otherwise be done on Web pages. However, there are now easier ways to do many of the more basic things that can be done with applets, and they are no longer the main focus of interest in Java. Nevertheless, there are still some things that can be done best with applets, and they are still somewhat common on the Web. We will look at applets in the next section.

A stand-alone application is a program that runs on its own, without depending on a Web browser. You've been writing stand-alone applications all along. Any class that has a `main()` routine defines a stand-alone application; running the program just means executing this `main()` routine. However, the programs that you've seen up till now have been "command-line" programs, where the user and computer interact by typing things back and forth to each

other. A GUI program offers a much richer type of user interface, where the user uses a mouse and keyboard to interact with GUI components such as windows, menus, buttons, check boxes, text input boxes, scroll bars, and so on. The main routine of a GUI program creates one or more such components and displays them on the computer screen. Very often, that's all it does. Once a GUI component has been created, it follows its **own** programming—programming that tells it how to draw itself on the screen and how to respond to events such as being clicked on by the user.

A GUI program doesn't have to be immensely complex. We can, for example, write a very simple GUI "Hello World" program that says "Hello" to the user, but does it by opening a window where the greeting is displayed:

```
import javax.swing.JOptionPane;

public class HelloWorldGUI1 {

   public static void main(String[] args) {
      JOptionPane.showMessageDialog( null, "Hello World!" );
   }

}
```

When this program is run, a window appears on the screen that contains the message "Hello World!". The window also contains an "OK" button for the user to click after reading the message. When the user clicks this button, the window closes and the program ends. By the way, this program can be placed in a file named *HelloWorldGUI1.java*, compiled, and run just like any other Java program.

Now, this program is already doing some pretty fancy stuff. It creates a window, it draws the contents of that window, and it handles the event that is generated when the user clicks the button. The reason the program was so easy to write is that all the work is done by `showMessageDialog()`, a `static` method in the built-in class *JOptionPane*. (Note that the source code "imports" the class `javax.swing.JOptionPane` to make it possible to refer to the *JOptionPane* class using its simple name. See Subsection 4.5.3 for information about importing classes from Java's standard packages.)

If you want to display a message to the user in a GUI program, this is a good way to do it: Just use a standard class that already knows how to do the work! And in fact, *JOptionPane* is regularly used for just this purpose (but as part of a larger program, usually). Of course, if you want to do anything serious in a GUI program, there is a lot more to learn. To give you an idea of the types of things that are involved, we'll look at a short GUI program that does the same things as the previous program—open a window containing a message and an OK button, and respond to a click on the button by ending the program—but does it all by hand instead of by using the built-in *JOptionPane* class. Mind you, this is **not** a good way to write the program, but it will illustrate some important aspects of GUI programming in Java.

Here is the source code for the program. You are not expected to understand it yet. I will explain how it works below, but it will take the rest of the chapter before you will really understand completely. In this section, you will just get a brief overview of GUI programming.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class HelloWorldGUI2 {

   private static class HelloWorldDisplay extends JPanel {
```

```
            public void paintComponent(Graphics g) {
               super.paintComponent(g);
               g.drawString( "Hello World!", 20, 30 );
            }
         }

         private static class ButtonHandler implements ActionListener {
            public void actionPerformed(ActionEvent e) {
               System.exit(0);
            }
         }

         public static void main(String[] args) {

            HelloWorldDisplay displayPanel = new HelloWorldDisplay();
            JButton okButton = new JButton("OK");
            ButtonHandler listener = new ButtonHandler();
            okButton.addActionListener(listener);

            JPanel content = new JPanel();
            content.setLayout(new BorderLayout());
            content.add(displayPanel, BorderLayout.CENTER);
            content.add(okButton, BorderLayout.SOUTH);

            JFrame window = new JFrame("GUI Test");
            window.setContentPane(content);
            window.setSize(250,100);
            window.setLocation(100,100);
            window.setVisible(true);

         }

      }
```

6.1.1 JFrame and JPanel

In a Java GUI program, each GUI component in the interface is represented by an object in the program. One of the most fundamental types of component is the ***window***. Windows have many behaviors. They can be opened and closed. They can be resized. They have "titles" that are displayed in the title bar above the window. And most important, they can contain other GUI components such as buttons and menus.

Java, of course, has a built-in class to represent windows. There are actually several different types of window, but the most common type is represented by the *JFrame* class (which is included in the package `javax.swing`). A *JFrame* is an independent window that can, for example, act as the main window of an application. One of the most important things to understand is that a *JFrame* object comes with many of the behaviors of windows already programmed in. In particular, it comes with the basic properties shared by all windows, such as a titlebar and the ability to be opened and closed. Since a *JFrame* comes with these behaviors, you don't have to program them yourself! This is, of course, one of the central ideas of object-oriented programming. What a *JFrame* doesn't come with, of course, is ***content***, the stuff that is contained in the window. If you don't add any other content to a *JFrame*, it will just display a blank area. You can add content either by creating a *JFrame* object and then adding the content to it or by creating a subclass of *JFrame* and adding the content in the constructor of that subclass.

The main program above declares a variable, `window`, of type *JFrame* and sets it to refer to a new window object with the statement:

```
JFrame window = new JFrame("GUI Test");
```

The parameter in the constructor, "GUI Test", specifies the title that will be displayed in the titlebar of the window. This line creates the window object, but the window itself is not yet visible on the screen. Before making the window visible, some of its properties are set with these statements:

```
window.setContentPane(content);
window.setSize(250,100);
window.setLocation(100,100);
```

The first line here sets the content of the window. (The content itself was created earlier in the main program.) The second line says that the window will be 250 pixels wide and 100 pixels high. The third line says that the upper left corner of the window will be 100 pixels over from the left edge of the screen and 100 pixels down from the top. Once all this has been set up, the window is actually made visible on the screen with the command:

```
window.setVisible(true);
```

It might look as if the program ends at that point, and, in fact, the `main()` routine does end. However, the window is still on the screen and the program as a whole does not end until the user clicks the OK button. Once the window was opened, a new thread was created to manage the graphical user interface, and that thread continues to run even after `main()` has finished.

* * *

The content that is displayed in a *JFrame* is called its ***content pane***. (In addition to its content pane, a *JFrame* can also have a menu bar, which is a separate thing that I will talk about later.) A basic *JFrame* already has a blank content pane; you can either add things to that pane or you can replace the basic content pane entirely. In my sample program, the line `window.setContentPane(content)` replaces the original blank content pane with a different component. (Remember that a "component" is just a visual element of a graphical user interface.) In this case, the new content is a component of type *JPanel*.

JPanel is another of the fundamental classes in Swing. The basic *JPanel* is, again, just a blank rectangle. There are two ways to make a useful *JPanel*: The first is to **add other components** to the panel; the second is to **draw something** in the panel. Both of these techniques are illustrated in the sample program. In fact, you will find two *JPanels* in the program: `content`, which is used to contain other components, and `displayPanel`, which is used as a drawing surface.

Let's look more closely at `displayPanel`. This variable is of type *HelloWorldDisplay*, which is a nested static class inside the *HelloWorldGUI2* class. (Nested classes were introduced in Subsection 5.7.2.) This class defines just one instance method, `paintComponent()`, which overrides a method of the same name in the *JPanel* class:

```
private static class HelloWorldDisplay extends JPanel {
   public void paintComponent(Graphics g) {
      super.paintComponent(g);
      g.drawString( "Hello World!", 20, 30 );
   }
}
```

The `paintComponent()` method is called by the system when a component needs to be painted on the screen. In the *JPanel* class, the `paintComponent` method simply fills the panel with the panel's background color. The `paintComponent()` method in *HelloWorldDisplay* begins by calling `super.paintComponent(g)`. This calls the version of `paintComponent()` that is defined in the superclass, *JPanel*; that is, it fills the panel with the background color. (See Subsection 5.6.2 for a discussion of the special variable `super`.) Then it calls `g.drawString()` to paint the string "Hello World!" onto the panel. The net result is that whenever a *HelloWorldDisplay* is shown on the screen, it displays the string "Hello World!".

We will often use *JPanels* in this way, as drawing surfaces. Usually, when we do this, we will define a nested class that is a subclass of *JPanel* and we will write a `paintComponent` method in that class to draw the desired content in the panel.

6.1.2 Components and Layout

Another way of using a *JPanel* is as a ***container*** to hold other components. Java has many classes that define GUI components. Before these components can appear on the screen, they must be ***added*** to a container. In this program, the variable named `content` refers to a *JPanel* that is used as a container, and two other components are added to that container. This is done in the statements:

```
content.add(displayPanel, BorderLayout.CENTER);
content.add(okButton, BorderLayout.SOUTH);
```

Here, `content` refers to an object of type *JPanel*; later in the program, this panel becomes the content pane of the window. The first component that is added to `content` is `displayPanel` which, as discussed above, displays the message, "Hello World!". The second is `okButton` which represents the button that the user clicks to close the window. The variable `okButton` is of type *JButton*, the Java class that represents push buttons.

The "BorderLayout" stuff in these statements has to do with how the two components are arranged in the container. When components are added to a container, there has to be some way of deciding how those components are arranged inside the container. This is called "laying out" the components in the container, and the most common technique for laying out components is to use a ***layout manager***. A layout manager is an object that implements some policy for how to arrange the components in a container; different types of layout manager implement different policies. One type of layout manager is defined by the *BorderLayout* class. In the program, the statement

```
content.setLayout(new BorderLayout());
```

creates a new *BorderLayout* object and tells the `content` panel to use the new object as its layout manager. Essentially, this line determines how components that are added to the content panel will be arranged inside the panel. We will cover layout managers in much more detail later, but for now all you need to know is that adding `okButton` in the `BorderLayout.SOUTH` position puts the button at the bottom of the panel, and putting `displayPanel` in the `BorderLayout.CENTER` position makes it fill any space that is not taken up by the button.

This example shows a general technique for setting up a GUI: Create a container and assign a layout manager to it, create components and add them to the container, and use the container as the content pane of a window or applet. A container is itself a component, so it is possible that some of the components that are added to the top-level container are themselves containers, with their own layout managers and components. This makes it possible to build up complex user interfaces in a hierarchical fashion, with containers inside containers inside containers...

6.1.3 Events and Listeners

The structure of containers and components sets up the physical appearance of a GUI, but it doesn't say anything about how the GUI **behaves**. That is, what can the user do to the GUI and how will it respond? GUIs are largely ***event-driven***; that is, the program waits for events that are generated by the user's actions (or by some other cause). When an event occurs, the program responds by executing an ***event-handling method***. In order to program the behavior of a GUI, you have to write event-handling methods to respond to the events that you are interested in.

The most common technique for handling events in Java is to use ***event listeners***. A listener is an object that includes one or more event-handling methods. When an event is detected by another object, such as a button or menu, the listener object is notified and it responds by running the appropriate event-handling method. An event is detected or generated by an object. Another object, the listener, has the responsibility of responding to the event. The event itself is actually represented by a third object, which carries information about the type of event, when it occurred, and so on. This division of responsibilities makes it easier to organize large programs.

As an example, consider the OK button in the sample program. When the user clicks the button, an event is generated. This event is represented by an object belonging to the class *ActionEvent*. The event that is generated is associated with the button; we say that the button is the ***source*** of the event. The listener object in this case is an object belonging to the class *ButtonHandler*, which is defined as a nested class inside *HelloWorldGUI2*:

```
private static class ButtonHandler implements ActionListener {
   public void actionPerformed(ActionEvent e) {
      System.exit(0);
   }
}
```

This class implements the *ActionListener* interface—a requirement for listener objects that handle events from buttons. (Interfaces were introduced in Subsection 5.7.1.) The event-handling method is named `actionPerformed`, as specified by the *ActionListener* interface. This method contains the code that is executed when the user clicks the button; in this case, the code is a call to `System.exit()`, which will terminate the program.

There is one more ingredient that is necessary to get the event from the button to the listener object: The listener object must ***register*** itself with the button as an event listener. This is done with the statement:

```
okButton.addActionListener(listener);
```

This statement tells `okButton` that when the user clicks the button, the ActionEvent that is generated should be sent to `listener`. Without this statement, the button has no way of knowing that some other object would like to listen for events from the button.

This example shows a general technique for programming the behavior of a GUI: Write classes that include event-handling methods. Create objects that belong to these classes and register them as listeners with the objects that will actually detect or generate the events. When an event occurs, the listener is notified, and the code that you wrote in one of its event-handling methods is executed. At first, this might seem like a very roundabout and complicated way to get things done, but as you gain experience with it, you will find that it is very flexible and that it goes together very well with object oriented programming. (We will return to events

and listeners in much more detail in Section 6.3 and later sections; I do not expect you to completely understand them at this time.)

6.2 Applets and HTML

ALTHOUGH STAND-ALONE APPLICATIONS are much more important than applets at this point in the history of Java, applets are still widely used. They can do things on Web pages that can't easily be done with other technologies. It is easy to distribute applets to users: The user just has to open a Web page, and the applet is there, with no special installation required (although the user must have an appropriate version of Java installed on their computer). And of course, applets are fun; now that the Web has become such a common part of life, it's nice to be able to see your work running on a web page.

The good news is that writing applets is not much different from writing stand-alone applications. The structure of an applet is essentially the same as the structure of the *JFrames* that were introduced in the previous section, and events are handled in the same way in both types of program. So, most of what you learn about applications applies to applets, and *vice versa*.

Of course, one difference is that an applet is dependent on a Web page, so to use applets effectively, you have to learn at least a little about creating Web pages. Web pages are written using a language called ***HTML*** (HyperText Markup Language). In Subsection 6.2.3, below, you'll learn how to use HTML to create Web pages that display applets.

6.2.1 JApplet

The *JApplet* class (in package `javax.swing`) can be used as a basis for writing applets in the same way that *JFrame* is used for writing stand-alone applications. The basic *JApplet* class represents a blank rectangular area. Since an applet is not a stand-alone application, this area must appear on a Web page, or in some other environment that knows how to display an applet. Like a *JFrame*, a *JApplet* contains a content pane (and can contain a menu bar). You can add content to an applet either by adding content to its content pane or by replacing the content pane with another component. In my examples, I will generally create a *JPanel* and use it as a replacement for the applet's content pane.

To create an applet, you will write a subclass of *JApplet*. The *JApplet* class defines several instance methods that are unique to applets. These methods are called by the applet's environment at certain points during the applet's "life cycle." In the *JApplet* class itself, these methods do nothing; you can override these methods in a subclass. The most important of these special applet methods is

```
public void init()
```

An applet's `init()` method is called when the applet is created. You can use the `init()` method as a place where you can set up the physical structure of the applet and the event handling that will determine its behavior. (You can also do some initialization in the constructor for your class, but there are certain aspects of the applet's environment that are set up after its constructor is called but before the `init()` method is called, so there are a few operations that will work in the `init()` method but will not work in the constructor.) The other applet life-cycle methods are `start()`, `stop()`, and `destroy()`. I will not use these methods for the time being and will not discuss them here except to mention that `destroy()` is called at the end of the applet's lifetime and can be used as a place to do any necessary cleanup, such as closing any windows that were opened by the applet.

With this in mind, we can look at our first example of a *JApplet*. It is, of course, an applet that says "Hello World!". To make it a little more interesting, I have added a button that changes the text of the message, and a state variable, `currentMessage`, that holds the text of the current message. This example is very similar to the stand-alone application *HelloWorldGUI2* from the previous section. It uses an event-handling class to respond when the user clicks the button, a panel to display the message, and another panel that serves as a container for the message panel and the button. The second panel becomes the content pane of the applet. Here is the source code for the applet; again, you are not expected to understand all the details at this time:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * A simple applet that can display the messages "Hello World"
 * and "Goodbye World".  The applet contains a button, and it
 * switches from one message to the other when the button is
 * clicked.
 */
public class HelloWorldApplet extends JApplet {

   private String currentMessage = "Hello World!"; // Currently displayed message.
   private MessageDisplay displayPanel;  // The panel where the message is displayed.

   private class MessageDisplay extends JPanel {   // Defines the display panel.
      public void paintComponent(Graphics g) {
         super.paintComponent(g);
         g.drawString(currentMessage, 20, 30);
      }
   }

   private class ButtonHandler implements ActionListener {  // The event listener.
      public void actionPerformed(ActionEvent e) {
         if (currentMessage.equals("Hello World!"))
            currentMessage = "Goodbye World!";
         else
            currentMessage = "Hello World!";
         displayPanel.repaint(); // Paint display panel with new message.
      }
   }

   /**
    * The applet's init() method creates the button and display panel and
    * adds them to the applet, and it sets up a listener to respond to
    * clicks on the button.
    */
   public void init() {

      displayPanel = new MessageDisplay();
      JButton changeMessageButton = new JButton("Change Message");
      ButtonHandler listener = new ButtonHandler();
      changeMessageButton.addActionListener(listener);

      JPanel content = new JPanel();
      content.setLayout(new BorderLayout());
```

```
        content.add(displayPanel, BorderLayout.CENTER);
        content.add(changeMessageButton, BorderLayout.SOUTH);

        setContentPane(content);
    }

}
```

You should compare this class with *HelloWorldGUI2.java* from the previous section. One subtle difference that you will notice is that the member variables and nested classes in this example are non-static. Remember that an applet is an object. A single class can be used to make several applets, and each of those applets will need its own copy of the applet data, so the member variables in which the data are stored must be non-static instance variables. Since the variables are non-static, the two nested classes, which use those variables, must also be non-static. (Static nested classes cannot access non-static member variables in the containing class; see Subsection 5.7.2.) Remember the basic rule for deciding whether to make a nested class static: If it needs access to any instance variable or instance method in the containing class, the nested class must be non-static; otherwise, it can be declared to be `static`.

(By the way, *JApplet* is a subclass of a more basic class, named *Applet* and found in the package `java.applet`. *JApplet* is part of the Swing GUI framework *Applet* is part of the older AWT and is no longer commonly used directly for writing applets.)

6.2.2 Reusing Your JPanels

Both applets and frames can be programmed in the same way: Design a *JPanel*, and use it to replace the default content pane in the applet or frame. This makes it very easy to write two versions of a program, one which runs as an applet and one which runs as a frame. The idea is to create a subclass of *JPanel* that represents the content pane for your program; all the hard programming work is done in this panel class. An object of this class can then be used as the content pane either in a frame or in an applet—or both. Only a very simple `main()` program is needed to show your panel in a frame, and only a very simple applet class is needed to show your panel in an applet, so it's easy to make both versions.

As an example, we can rewrite *HelloWorldApplet* by writing a subclass of *JPanel*. That class can then be reused to make a frame in a standalone application. This class is very similar to *HelloWorldApplet*, but now the initialization is done in a constructor instead of in an `init()` method:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class HelloWorldPanel extends JPanel {

   private String currentMessage = "Hello World!"; // Currently displayed message.
   private MessageDisplay displayPanel;  // The panel where the message is displayed.

   private class MessageDisplay extends JPanel {   // Defines the display panel.
      public void paintComponent(Graphics g) {
         super.paintComponent(g);
         g.drawString(currentMessage, 20, 30);
      }
   }

   private class ButtonHandler implements ActionListener {  // The event listener.
```

```
            public void actionPerformed(ActionEvent e) {
               if (currentMessage.equals("Hello World!"))
                  currentMessage = "Goodbye World!";
               else
                  currentMessage = "Hello World!";
               displayPanel.repaint(); // Paint display panel with new message.
            }
         }

         /**
          * The constructor creates the components that will be contained inside this
          * panel, and then adds those components to this panel.
          */
         public HelloWorldPanel() {

            displayPanel = new MessageDisplay();  // Create the display subpanel.

            JButton changeMessageButton = new JButton("Change Message"); // The button.
            ButtonHandler listener = new ButtonHandler();
            changeMessageButton.addActionListener(listener);

            setLayout(new BorderLayout());  // Set the layout manager for this panel.
            add(displayPanel, BorderLayout.CENTER);  // Add the display panel.
            add(changeMessageButton, BorderLayout.SOUTH);  // Add the button.

         }

      }
```

Once this class exists, it can be used in an applet. The applet class only has to create an object of type *HelloWorldPanel* and use that object as its content pane:

```
import javax.swing.JApplet;

public class HelloWorldApplet2 extends JApplet {
   public void init() {
      HelloWorldPanel content = new HelloWorldPanel();
      setContentPane(content);
   }
}
```

Similarly, it's easy to make a frame that uses an object of type *HelloWorldPanel* as its content pane:

```
import javax.swing.JFrame;

public class HelloWorldGUI3 {

   public static void main(String[] args) {
      JFrame window = new JFrame("GUI Test");
      HelloWorldPanel content = new HelloWorldPanel();
      window.setContentPane(content);
      window.setSize(250,100);
      window.setLocation(100,100);
      window.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      window.setVisible(true);
   }

}
```

One new feature of this example is the line

```
window.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
```

This says that when the user closes the window by clicking the close box in the title bar of the window, the program should be terminated. This is necessary because no other way is provided to end the program. Without this line, the default close operation of the window would simply hide the window when the user clicks the close box, leaving the program running even though nothing is visible on the screen. This brings up one of the difficulties of reusing the same panel class both in an applet and in a frame: There are some things that a stand-alone application can do that an applet can't do. Terminating the program is one of those things. If an applet calls `System.exit()`, it has no effect except to generate an error.

Nevertheless, in spite of occasional minor difficulties, many of the GUI examples in this book will be written as subclasses of *JPanel* that can be used either in an applet or in a frame.

6.2.3 Basic HTML

Before you can actually use an applet that you have written, you need to create a Web page on which to place the applet. Such pages are themselves written in a language called ***HTML*** (HyperText Markup Language). An HTML document describes the contents of a page. A Web browser interprets the HTML code to determine what to display on the page. The HTML code doesn't look much like the resulting page that appears in the browser. The HTML document does contain all the text that appears on the page, but that text is "marked up" with commands that determine the structure and appearance of the text and determine what will appear on the page in addition to the text.

HTML has become a rather complicated language, and it is only one of the languages that you need to be familiar with if you want to write sophisticated modern web pages. Many aspects of the visual style of a page can be controlled using a language called CSS (cascading style sheets). Web pages can be dynamic and interactive, and their behavior can be programmed using a programming language called JavaScript (which is only very distantly related to Java). Furthermore, interactive web pages often work with programs that run on the Web server, which can be written in Java or in several other languages. Programming for the web has become very complicated indeed!

Nevertheless, it's fairly easy to write basic web pages using only plain HTML. In this section, I will cover just the most basic aspects of the language. You can easily find more information on the Web, if you want to learn more. Although there are many Web-authoring programs that make it possible to create Web pages without ever looking at the underlying HTML code, it is possible to write an HTML page using an ordinary text editor, typing in all the mark-up commands by hand, and it is worthwhile to learn how to create at least simple pages in this way.

There is a strict syntax for HTML documents (although in practice Web browsers will do their best to display a page even if it does not follow the syntax strictly). Leaving out optional features, an HTML document has the form:

```
<html>
<head>
<title>⟨document-title⟩</title>
</head>
<body>
⟨document-content⟩
```

```
</body>
</html>
```

The ⟨*document-title*⟩ is text that will appear in the title bar of the Web browser window when the page is displayed. The ⟨*document-content*⟩ is what is displayed on the page itself. The rest of this section describes some of the things that can go into the ⟨*document-content*⟩ section of an HTML document.

* * *

The mark-up commands used by HTML are called ***tags***. Examples include `<html>` and `<title>` in the document outline given above. An HTML tag takes the form

```
<⟨tag-name⟩  ⟨optional-modifiers⟩>
```

where the ⟨*tag-name*⟩ is a word that specifies the command, and the ⟨*optional-modifiers*⟩, if present, are used to provide additional information for the command (much like parameters in subroutines). A modifier takes the form

```
⟨modifier-name⟩ = ⟨value⟩
```

Usually, the ⟨*value*⟩ is enclosed in quotes, and it must be if it is more than one word long or if it contains certain special characters. There are a few modifiers which have no value, in which case only the name of the modifier is present. HTML is case insensitive, which means that you can use upper case and lower case letters interchangeably in tags and modifiers. (However, lower case is generally used because XHTML, a successor language to HTML, requires lower case.)

A simple example of a tag is `<hr>`, which draws a line—also called a "horizontal rule"—across the page. The `hr` tag can take several possible modifiers such as `width` and `align`. For example, a horizontal line that extends halfway across the page could be generated with the tag:

```
<hr  width="50%">
```

The `width` here is specified as `50%` of the available space, meaning a line that extends halfway across the page. The width could also be given as a fixed number of pixels.

Many tags require matching closing tags, which take the form

```
</⟨tag-name⟩>
```

For example, the `<html>` tag at the beginning of an HTML document must be matched by a closing `</html>` tag at the end of the document. As another example, the tag `<pre>` must always have a matching closing tag `</pre>` later in the document. An opening/closing tag pair applies to everything that comes between the opening tag and the closing tag. The `<pre>` tag tells a Web browser to display everything between the `<pre>` and the `</pre>` just as it is formatted in the original HTML source code, including all the spaces and carriage returns. (But tags between `<pre>` and `</pre>` are still interpreted by the browser.) "Pre" stands for preformatted text. All of the sample programs in the on-line version of this book are formatted using the `<pre>` command.

It is important for you to understand that when you don't use `<pre>`, the computer will completely ignore the formatting of the text in the HTML source code. The only thing it pays attention to is the tags. Five blank lines in the source code have no more effect than one blank line or even a single blank space. Outside of `<pre>`, if you want to force a new line on the Web page, you can use the tag `<br>`, which stands for "break". For example, I might give my address as:

```
David Eck<br>
Department of Mathematics and Computer Science<br>
Hobart and William Smith Colleges<br>
Geneva, NY 14456<br>
```

If you want extra vertical space in your web page, you can use several `<br>`'s in a row.

Similarly, you need a tag to indicate how the text should be broken up into paragraphs. This is done with the `<p>` tag, which should be placed at the beginning of every paragraph. The `<p>` tag has a matching `</p>`, which should be placed at the end of each paragraph. The closing `</p>` is technically optional, but it is considered good form to use it. If you want all the lines of the paragraph to be shoved over to the right, you can use `<p align=right>` instead of `<p>`. (This is mostly useful when used with one short line, or when used with `<br>` to make several short lines.) You can also use `<p align=center>` for centered lines.

By the way, if tags like `<p>` and `<hr>` have special meanings in HTML, you might wonder how to get them to appear literally on a web page. To get certain special characters to appear on the page, you have to use an ***entity name*** in the HTML source code. The entity name for `<` is `&lt;`, and the entity name for `>` is `&gt;`. Entity names begin with `&` and end with a semicolon. The character `&` is itself a special character whose entity name is `&amp;`. There are also entity names for nonstandard characters such as an accented "e", which has the entity name `&eacute;` and the Greek letter π, which is written as `&pi;`.

There are several useful tags that change the appearance of text. To get italic text, enclose the text between `<i>` and `</i>`. For example,

```
<i>Introduction to Programming using Java</i>
```

in an HTML document gives *Introduction to Programming using Java* in italics when the document is displayed as a Web page. The tags `<b>`, `<u>`, and `<tt>` can be used in a similar way for **bold**, underlined, and `typewriter-style` ("monospace") text.

A headline, with very large text, can be made by placing the text between `<h1>` and `</h1>`. Headlines with smaller text can be made using `<h2>` or `<h3>` instead of `<h1>`. Note that these headline tags stand on their own; they are not used inside paragraphs. You can add the modifier `align=center` to center the headline, and you can right-justify it with `align=right`. You can include break tags (`<br>`) in a headline to break it up into multiple lines. For example, the following HTML code will produce a medium–sized, centered, two-line headline:

```
<h2 align=center>Chapter 6:<br>Introduction to GUI Programming</h2>
```

* * *

The most distinctive feature of HTML is that documents can contain ***links*** to other documents. The user can follow links from page to page and in the process visit pages from all over the Internet.

The `<a>` tag is used to create a link. The text between the `<a>` and its matching `</a>` appears on the page as the text of the link; the user can follow the link by clicking on this text. The `<a>` tag uses the modifier `href` to say which document the link should connect to. The value for `href` must be a ***URL*** (Uniform Resource Locator). A URL is a coded set of instructions for finding a document on the Internet. For example, the URL for my own "home page" is

```
http://math.hws.edu/eck/
```

To make a link to this page, I would use the HTML source code

```
<a href="http://math.hws.edu/eck/">David's Home Page</a>
```

The best place to find URLs is on existing Web pages. Web browsers display the URL for the page you are currently viewing, and many browsers will display the URL of a link if you point to the link with the mouse.

If you are writing an HTML document and you want to make a link to another document that is in the same directory, you can use a ***relative URL***. The relative URL consists of just the name of the file. For example, to create a link to a file named "s1.html" in the same directory as the HTML document that you are writing, you could use

```
<a href="s1.html">Section 1</a>
```

There are also relative URLs for linking to files that are in other directories. Using relative URLs is a good idea, since if you use them, you can move a whole collection of files without changing any of the links between them (as long as you don't change the relative locations of the files).

When you type a URL into a Web browser, you can omit the "http://" at the beginning of the URL. However, in an `<a>` tag in an HTML document, the "http://" can only be omitted if the URL is a relative URL. For a normal URL, it is required.

* * *

You can add images to a Web page with the `<img>` tag. (This is a tag that has no matching closing tag.) The actual image must be stored in a separate file from the HTML document. The `<img>` tag has a required modifier, named `src`, to specify the URL of the image file. For most browsers, the image should be in one of the formats PNG (with a file name ending in ".png"), JPEG (with a file name ending in ".jpeg" or ".jpg"), or GIF (with a file name ending in ".gif"). Usually, the image is stored in the same place as the HTML document, and a relative URL—that is, just the name of the file—is used to specify the image file.

The `<img>` tag also has several optional modifiers. It's a good idea to always include the `height` and `width` modifiers, which specify the size of the image in pixels. Some browsers handle images better if they know in advance how big they are. The `align` modifier can be used to affect the placement of the image: "align=right" will shove the image to the right edge of the page, and the text on the page will flow around the image; "align=left" works similarly. (Unfortunately, "align=center" doesn't have the meaning you would expect. Browsers treat images as if they are just big characters. Images can occur inside paragraphs, links, and headings, for example. Alignment values of `center`, `top`, and `bottom` are used to specify how the image should line up with other characters in a line of text: Should the baseline of the text be at the center, the top, or the bottom of the image? Alignment values of `right` and `left` were added to HTML later, but they are the most useful values. If you want an image centered on the page, put it inside a `<p align=center>` tag.)

For example, here is HTML code that will place an image from a file named figure1.png on the page.

```
<img src="figure1.png" align=right height=150 width=100>
```

The image is 100 pixels wide and 150 pixels high, and it will appear on the right edge of the page.

6.2.4 Applets on Web Pages

The main point of this whole discussion of HTML is to learn how to use applets on the Web. The `<applet>` tag can be used to add a Java applet to a Web page. This tag must have a matching `</applet>`. A required modifier named `code` gives the name of the compiled class

file that contains the applet class. The modifiers `height` and `width` are required to specify the size of the applet, in pixels. If you want the applet to be centered on the page, you can put the applet in a paragraph with `center` alignment. So, an applet tag to display an applet named `HelloWorldApplet` centered on a Web page would look like this:

```
<p align=center>
<applet code="HelloWorldApplet.class" height=100 width=250>
</applet>
</p>
```

This assumes that the file `HelloWorldApplet.class` is located in the same directory with the HTML document. If this is not the case, you can use another modifier, `codebase`, to give the URL of the directory that contains the class file. The value of `code` itself is always just a class, not a URL.

If the applet uses other classes in addition to the applet class itself, then those class files must be in the same directory as the applet class (always assuming that your classes are all in the "default package"; see Subsection 2.6.4; if not, they must be in subdirectories). If an applet requires more than one or two class files, it's a good idea to collect all the class files into a single jar file. Jar files are "archive files" which hold a number of smaller files. If your class files are in a jar archive, then you have to specify the name of the jar file in an `archive` modifier in the `<applet>` tag, as in

```
<applet code="HelloWorldApplet.class" archive="HelloWorld.jar" height=50...
```

I will have more to say about creating and using jar files at the end of this chapter.

Applets can use ***applet parameters*** to customize their behavior. Applet parameters are specified by using `<param>` tags, which can only occur between an `<applet>` tag and the closing `</applet>`. The `param` tag has required modifiers named `name` and `value`, and it takes the form

```
<param  name="⟨param-name⟩"  value="⟨param-value⟩">
```

The parameters are available to the applet when it runs. An applet uses the predefined method `getParameter()` to check for parameters specified in `param` tags. The `getParameter()` method has the following interface:

```
String getParameter(String paramName)
```

The parameter `paramName` corresponds to the ⟨*param-name*⟩ in a `param` tag. If the specified `paramName` actually occurs in one of the `param` tags, then `getParameter(paramName)` returns the associated ⟨*param-value*⟩. If the specified `paramName` does not occur in any `param` tag, then `getParameter(paramName)` returns the value `null`. Parameter names are case-sensitive, so you cannot use "size" in the `param` tag and ask for "Size" in `getParameter`. The `getParameter()` method is often called in the applet's `init()` method. It will not work correctly in the applet's constructor, since it depends on information about the applet's environment that is not available when the constructor is called.

Here is an example of an `applet` tag with several `params`:

```
<applet code="ShowMessage.class" width=200 height=50>
   <param name="message" value="Goodbye World!">
   <param name="font" value="Serif">
   <param name="size" value="36">
</applet>
```

The `ShowMessage` applet would presumably read these parameters in its `init()` method, which could go something like this:

```
String message;  // Instance variable: message to be displayed.
String fontName; // Instance variable: font to use for display.
int fontSize;    // Instance variable: size of the display font.

public void init() {
    String value;
    value = getParameter("message"); // Get message param, if any.
    if (value == null)
       message = "Hello World!";  // Default value, if no param is present.
    else
       message = value;  // Value from PARAM tag.
    value = getParameter("font");
    if (value == null)
       fontName = "SansSerif";  // Default value, if no param is present.
    else
       fontName = value;
    value = getParameter("size");
    try {
       fontSize = Integer.parseInt(value);  // Convert string to number.
    }
    catch (NumberFormatException e) {
       fontSize = 20; // Default value, if no param is present, or if
    }                 //   the parameter value is not a legal integer.
     .
     .
     .
```

Elsewhere in the applet, the instance variables `message`, `fontName`, and `fontSize` would be used to determine the message displayed by the applet and the appearance of that message. Note that the value returned by `getParameter()` is always a *String*. If the `param` represents a numerical value, the string must be converted into a number, as is done here for the `size` parameter.

6.3 Graphics and Painting

EVERYTHING YOU SEE ON A COMPUTER SCREEN has to be drawn there, even the text. The Java API includes a range of classes and methods that are devoted to drawing. In this section, I'll look at some of the most basic of these.

The physical structure of a GUI is built of components. The term ***component*** refers to a visual element in a GUI, including buttons, menus, text-input boxes, scroll bars, check boxes, and so on. In Java, GUI components are represented by objects belonging to subclasses of the class `java.awt.Component`. Most components in the Swing GUI—although not top-level components like JApplet and JFrame—belong to subclasses of the class `javax.swing.JComponent`, which is itself a subclass of `java.awt.Component`. Every component is responsible for drawing itself. If you want to use a standard component, you only have to add it to your applet or frame. You don't have to worry about painting it on the screen. That will happen automatically, since it already knows how to draw itself.

Sometimes, however, you do want to draw on a component. You will have to do this whenever you want to display something that is not included among the standard, pre-defined

component classes. When you want to do this, you have to define your own component class and provide a method in that class for drawing the component. I will always use a subclass of *JPanel* when I need a drawing surface of this kind, as I did for the *MessageDisplay* class in the example *HelloWorldApplet.java* in the previous section. A JPanel, like any JComponent, draws its content in the method

```
public void paintComponent(Graphics g)
```

To create a drawing surface, you should define a subclass of `JPanel` and provide a custom `paintComponent()` method. Create an object belonging to this class and use it in your applet or frame. When the time comes for your component to be drawn on the screen, the system will call its `paintComponent()` to do the drawing. That is, the code that you put into the `paintComponent()` method will be executed whenever the panel needs to be drawn on the screen; by writing this method, you determine the picture that will be displayed in the panel.

Note that the `paintComponent()` method has a parameter of type *Graphics*. The *Graphics* object will be provided by the system when it calls your method. You need this object to do the actual drawing. To do any drawing at all in Java, you need a ***graphics context***. A graphics context is an object belonging to the class `java.awt.Graphics`. Instance methods are provided in this class for drawing shapes, text, and images. Any given *Graphics* object can draw to only one location. In this chapter, that location will always be a GUI component belonging to some subclass of `JPanel`. The *Graphics* class is an abstract class, which means that it is impossible to create a graphics context directly, with a constructor. There are actually two ways to get a graphics context for drawing on a component: First of all, of course, when the `paintComponent()` method of a component is called by the system, the parameter to that method is a graphics context for drawing on the component. Second, every component has an instance method called `getGraphics()`. This method is a function that returns a graphics context that can be used for drawing on the component outside its `paintComponent()` method. The official line is that you should **not** do this, and I will almost always avoid it. But I have found it convenient to use `getGraphics()` in a few examples.

The `paintComponent()` method in the *JPanel* class simply fills the panel with the panel's background color. When defining a subclass of *JPanel* for use as a drawing surface, you will usually want to fill the panel with the background color before drawing other content onto the panel (although it is not necessary to do this if the drawing commands in the method cover the background of the component completely). This is traditionally done with a call to `super.paintComponent(g)`, so most `paintComponent()` methods that you write will have the form:

```
public void paintComponent(g) {
   super.paintComponent(g);
   . . . // Draw the content of the component.
}
```

* * *

Most components do, in fact, do all drawing operations in their `paintComponent()` methods. What happens if, in the middle of some other method, you realize that the content of the component needs to be changed? You should **not** call `paintComponent()` directly to make the change; this method is meant to be called only by the system. Instead, you have to inform the system that the component needs to be redrawn, and let the system do its job by calling `paintComponent()`. You do this by calling the component's `repaint()` method. The method

```
public void repaint();
```

is defined in the `Component` class, and so can be used with any component. You should call `repaint()` to inform the system that the component needs to be redrawn. It is important to understand that the `repaint()` method returns immediately, without doing any painting itself. The system will call the component's `paintComponent()` method *later*, as soon as it gets a chance to do so, after processing other pending events if there are any.

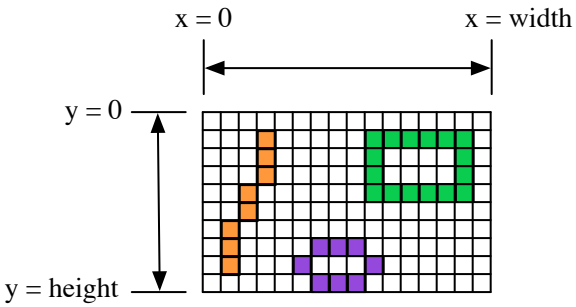
Note that the system can also call `paintComponent()` for other reasons. It is called when the component first appears on the screen. It will also be called if the size of the component changes, which can happen when the user resizes the window that contains the component. In versions of Java earlier than Java 6, `paintComponent()` is also called if the component is covered up and then uncovered, since the system did not automatically save a copy of the content. (And even in Java 6, the content is not automatically saved if is drawn with a graphics context created by `getGraphics()`, as I will do in some examples.) In any case, `paintComponent()` should be capable of redrawing the content of the component on demand. As you will see, however, some of our early examples will not be able to do this correctly.

This means that, to work properly, the `paintComponent()` method must be smart enough to correctly redraw the component at any time. To make this possible, a program should store data in its instance variables about the state of the component. These variables should contain all the information necessary to redraw the component completely. The `paintComponent()` method should use the data in these variables to decide what to draw. When the program wants to change the content of the component, it should not simply draw the new content. It should change the values of the relevant variables and call `repaint()`. When the system calls `paintComponent()`, that method will use the new values of the variables and will draw the component with the desired modifications. This might seem a roundabout way of doing things. Why not just draw the modifications directly? There are at least two reasons. First of all, it really does turn out to be easier to get things right if all drawing is done in one method. Second, even if you do make modifications directly, you still have to make the `paintComponent()` method aware of them in some way so that it will be able to **redraw** the component correctly on demand.

You will see how all this works in practice as we work through examples in the rest of this chapter. For now, we will spend the rest of this section looking at how to get some actual drawing done.

6.3.1 Coordinates

The screen of a computer is a grid of little squares called ***pixels***. The color of each pixel can be set individually, and drawing on the screen just means setting the colors of individual pixels.



A graphics context draws in a rectangle made up of pixels. A position in the rectangle is specified by a pair of integer coordinates, `(x,y)`. The upper left corner has coordinates `(0,0)`. The `x` coordinate increases from left to right, and the `y` coordinate increases from top to bottom. The illustration shows a 16-by-10 pixel component (with very large pixels). A small line, rectangle, and oval are shown as they would be drawn by coloring individual pixels. (Note that, properly speaking, the coordinates don't belong to the pixels but to the grid lines between them.)

For any component, you can find out the size of the rectangle that it occupies by calling the instance methods `getWidth()` and `getHeight()`, which return the number of pixels in the horizontal and vertical directions, respectively. In general, it's not a good idea to assume that you know the size of a component, since the size is often set by a layout manager and can even change if the component is in a window and that window is resized by the user. This means that it's good form to check the size of a component before doing any drawing on that component. For example, you can use a `paintComponent()` method that looks like:

```
public void paintComponent(Graphics g) {
   super.paintComponent(g);
   int width =  getWidth();   // Find out the width of this component.
   int height = getHeight();  // Find out its height.
   . . .   // Draw the content of the component.
}
```

Of course, your drawing commands will have to take the size into account. That is, they will have to use `(x,y)` coordinates that are calculated based on the actual height and width of the component.

6.3.2 Colors

You will probably want to use some color when you draw. Java is designed to work with the ***RGB color system***. An RGB color is specified by three numbers that give the level of red, green, and blue, respectively, in the color. A color in Java is an object of the class, `java.awt.Color`. You can construct a new color by specifying its red, blue, and green components. For example,

```
Color myColor = new Color(r,g,b);
```

There are two constructors that you can call in this way. In the one that I almost always use, `r`, `g`, and `b` are integers in the range 0 to 255. In the other, they are numbers of type **float** in the range 0.0F to 1.0F. (Recall that a literal of type **float** is written with an "F" to distinguish it from a **double** number.) Often, you can avoid constructing new colors altogether, since the *Color* class defines several named constants representing common colors: `Color.WHITE`, `Color.BLACK`, `Color.RED`, `Color.GREEN`, `Color.BLUE`, `Color.CYAN`, `Color.MAGENTA`, `Color.YELLOW`, `Color.PINK`, `Color.ORANGE`, `Color.LIGHT_GRAY`, `Color.GRAY`, and `Color.DARK_GRAY`. (There are older, alternative names for these constants that use lower case rather than upper case constants, such as `Color.red` instead of `Color.RED`, but the upper case versions are preferred because they follow the convention that constant names should be upper case.)

An alternative to RGB is the ***HSB color system***. In the HSB system, a color is specified by three numbers called the ***hue***, the ***saturation***, and the ***brightness***. The hue is the basic color, ranging from red through orange through all the other colors of the rainbow. The brightness is pretty much what it sounds like. A fully saturated color is a pure color tone. Decreasing the

saturation is like mixing white or gray paint into the pure color. In Java, the hue, saturation and brightness are always specified by values of type **float** in the range from 0.0F to 1.0F. The *Color* class has a `static` member function named `getHSBColor` for creating HSB colors. To create the color with HSB values given by `h`, `s`, and `b`, you can say:

```
Color myColor = Color.getHSBColor(h,s,b);
```

For example, to make a color with a random hue that is as bright and as saturated as possible, you could use:

```
Color randomColor = Color.getHSBColor( (float)Math.random(), 1.0F, 1.0F );
```

The type cast is necessary because the value returned by `Math.random()` is of type **double**, and `Color.getHSBColor()` requires values of type **float**. (By the way, you might ask why RGB colors are created using a constructor while HSB colors are created using a static member function. The problem is that we would need two different constructors, both of them with three parameters of type **float**. Unfortunately, this is impossible. You can have two constructors only if the number of parameters or the parameter types differ.)

The RGB system and the HSB system are just different ways of describing the same set of colors. It is possible to translate between one system and the other. The best way to understand the color systems is to experiment with them. In the on-line version of this section, you will find an applet that you can use to experiment with RGB and HSB colors.

One of the properties of a *Graphics* object is the current drawing color, which is used for all drawing of shapes and text. If `g` is a graphics context, you can change the current drawing color for `g` using the method `g.setColor(c)`, where `c` is a *Color*. For example, if you want to draw in green, you would just say `g.setColor(Color.GREEN)` before doing the drawing. The graphics context continues to use the color until you explicitly change it with another `setColor()` command. If you want to know what the current drawing color is, you can call the function `g.getColor()`, which returns an object of type *Color*. This can be useful if you want to change to another drawing color temporarily and then restore the previous drawing color.

Every component has an associated ***foreground color*** and ***background color***. Generally, the component is filled with the background color before anything else is drawn (although some components are "transparent," meaning that the background color is ignored). When a new graphics context is created for a component, the current drawing color is set to the foreground color. Note that the foreground color and background color are properties of the component, not of a graphics context.

The foreground and background colors can be set by instance methods `setForeground(c)` and `setBackground(c)`, which are defined in the `Component` class and therefore are available for use with any component. This can be useful even for standard components, if you want them to use colors that are different from the defaults.

6.3.3 Fonts

A ***font*** represents a particular size and style of text. The same character will appear different in different fonts. In Java, a font is characterized by a font name, a style, and a size. The available font names are system dependent, but you can always use the following four strings as font names: "Serif", "SansSerif", "Monospaced", and "Dialog". (A "serif" is a little decoration on a character, such as a short horizontal line at the bottom of the letter i. "SansSerif" means "without serifs." "Monospaced" means that all the characters in the font have the same width. The "Dialog" font is the one that is typically used in dialog boxes.)

The style of a font is specified using named constants that are defined in the *Font* class. You can specify the style as one of the four values:

- `Font.PLAIN`,
- `Font.ITALIC`,
- `Font.BOLD`, or
- `Font.BOLD + Font.ITALIC`.

The size of a font is an integer. Size typically ranges from about 10 to 36, although larger sizes can also be used. The size of a font is usually about equal to the height of the largest characters in the font, in pixels, but this is not an exact rule. The size of the default font is 12.

Java uses the class named `java.awt.Font` for representing fonts. You can construct a new font by specifying its font name, style, and size in a constructor:

```
Font plainFont = new Font("Serif", Font.PLAIN, 12);
Font bigBoldFont = new Font("SansSerif", Font.BOLD, 24);
```

Every graphics context has a current font, which is used for drawing text. You can change the current font with the `setFont()` method. For example, if `g` is a graphics context and `bigBoldFont` is a font, then the command `g.setFont(bigBoldFont)` will set the current font of `g` to `bigBoldFont`. The new font will be used for any text that is drawn *after* the `setFont()` command is given. You can find out the current font of `g` by calling the method `g.getFont()`, which returns an object of type *Font*.

Every component has an associated font. It can be set with the instance method `setFont(font)`, which is defined in the `Component` class. When a graphics context is created for drawing on a component, the graphic context's current font is set equal to the font of the component.

6.3.4 Shapes

The *Graphics* class includes a large number of instance methods for drawing various shapes, such as lines, rectangles, and ovals. The shapes are specified using the `(x,y)` coordinate system described above. They are drawn in the current drawing color of the graphics context. The current drawing color is set to the foreground color of the component when the graphics context is created, but it can be changed at any time using the `setColor()` method.

Here is a list of some of the most important drawing methods. With all these commands, any drawing that is done outside the boundaries of the component is ignored. Note that all these methods are in the *Graphics* class, so they all must be called through an object of type *Graphics*.

- `drawString(String str, int x, int y)` — Draws the text given by the string `str`. The string is drawn using the current color and font of the graphics context. `x` specifies the position of the left end of the string. `y` is the y-coordinate of the baseline of the string. The baseline is a horizontal line on which the characters rest. Some parts of the characters, such as the tail on a y or g, extend below the baseline.
- `drawLine(int x1, int y1, int x2, int y2)` — Draws a line from the point `(x1,y1)` to the point `(x2,y2)`. The line is drawn as if with a pen that hangs one pixel to the right and one pixel down from the `(x,y)` point where the pen is located. For example, if `g` refers to an object of type *Graphics*, then the command `g.drawLine(x,y,x,y)`, which

corresponds to putting the pen down at a point, colors the single pixel with upper left corner at the point `(x,y)`.

- `drawRect(int x, int y, int width, int height)` — Draws the outline of a rectangle. The upper left corner is at `(x,y)`, and the width and height of the rectangle are as specified. If `width` equals `height`, then the rectangle is a square. If the `width` or the `height` is negative, then nothing is drawn. The rectangle is drawn with the same pen that is used for `drawLine()`. This means that the actual width of the rectangle as drawn is `width+1`, and similarly for the height. There is an extra pixel along the right edge and the bottom edge. For example, if you want to draw a rectangle around the edges of the component, you can say "`g.drawRect(0, 0, getWidth()-1, getHeight()-1);`", where `g` is a graphics context for the component. If you use "`g.drawRect(0, 0, getWidth(), getHeight());`", then the right and bottom edges of the rectangle will be drawn *outside* the component and will not appear on the screen.
- `drawOval(int x, int y, int width, int height)` — Draws the outline of an oval. The oval is one that just fits inside the rectangle specified by `x`, `y`, `width`, and `height`. If `width` equals `height`, the oval is a circle.
- `drawRoundRect(int x, int y, int width, int height, int xdiam, int ydiam)` — Draws the outline of a rectangle with rounded corners. The basic rectangle is specified by `x`, `y`, `width`, and `height`, but the corners are rounded. The degree of rounding is given by `xdiam` and `ydiam`. The corners are arcs of an ellipse with horizontal diameter `xdiam` and vertical diameter `ydiam`. A typical value for `xdiam` and `ydiam` is 16, but the value used should really depend on how big the rectangle is.
- `draw3DRect(int x, int y, int width, int height, boolean raised)` — Draws the outline of a rectangle that is supposed to have a three-dimensional effect, as if it is raised from the screen or pushed into the screen. The basic rectangle is specified by `x`, `y`, `width`, and `height`. The `raised` parameter tells whether the rectangle seems to be raised from the screen or pushed into it. The 3D effect is achieved by using brighter and darker versions of the drawing color for different edges of the rectangle. The documentation recommends setting the drawing color equal to the background color before using this method. The effect won't work well for some colors.
- `drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)` — Draws part of the oval that just fits inside the rectangle specified by `x`, `y`, `width`, and `height`. The part drawn is an arc that extends `arcAngle` degrees from a starting angle at `startAngle` degrees. Angles are measured with 0 degrees at the 3 o'clock position (the positive direction of the horizontal axis). Positive angles are measured counterclockwise from zero, and negative angles are measured clockwise. To get an arc of a circle, make sure that `width` is equal to `height`.
- `fillRect(int x, int y, int width, int height)` — Draws a filled-in rectangle. This fills in the interior of the rectangle that would be drawn by `drawRect(x,y,width,height)`. The extra pixel along the bottom and right edges is not included. The `width` and `height` parameters give the exact width and height of the rectangle. For example, if you wanted to fill in the entire component, you could say "`g.fillRect(0, 0, getWidth(), getHeight());`"
- `fillOval(int x, int y, int width, int height)` — Draws a filled-in oval.
- `fillRoundRect(int x, int y, int width, int height, int xdiam, int ydiam)` — Draws a filled-in rounded rectangle.

- `fill3DRect(int x, int y, int width, int height, boolean raised)` — Draws a filled-in three-dimensional rectangle.
- `fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)` — Draw a filled-in arc. This looks like a wedge of pie, whose crust is the arc that would be drawn by the `drawArc` method.

6.3.5 Graphics2D

All drawing in Java is done through an object of type *Graphics*. The *Graphics* class provides basic commands for such things as drawing shapes and text and for selecting a drawing color. These commands are adequate in many cases, but they fall far short of what's needed in a serious computer graphics program. Java has another class, *Graphics2D*, that provides a larger set of drawing operations. *Graphics2D* is a sub-class of *Graphics*, so all the methods from the *Graphics* class are also available in a *Graphics2D*.

The `paintComponent()` method of a `JComponent` gives you a graphics context of type *Graphics* that you can use for drawing on the component. In fact, the graphics context actually belongs to the sub-class *Graphics2D* (in Java version 1.2 and later), and can be type-cast to gain access to the advanced *Graphics2D* drawing methods:

```
public void paintComponent(Graphics g) {
   super.paintComponent(g);
   Graphics2D g2;
   g2 = (Graphics2D)g;
    .
    . // Draw on the component using g2.
    .
}
```

Drawing in *Graphics2D* is based on shapes, which are objects that implement an interface named *Shape*. Shape classes include *Line2D*, *Rectangle2D*, *Ellipse2D*, *Arc2D*, and *GeneralPath*, among others; all these classes are defined in the package `java.awt.geom`. *Graphics2D* has methods `draw(Shape)` and `fill(Shape)` for drawing the outline of a shape and for filling its interior. Advanced capabilities include: lines that are more than one pixel thick, dotted and dashed lines, filling a shape with a texture (that is, with a repeated image), filling a shape with a gradient, and so-called "anti-aliased" drawing (which cuts down on the jagged appearance along a slanted line or curve).

In the *Graphics* class, coordinates are specified as integers and are based on pixels. The shapes that are used with *Graphics2D* use real numbers for coordinates, and they are not necessarily bound to pixels. In fact, you can change the coordinate system and use any coordinates that are convenient to your application. In computer graphics terms, you can apply a "transformation" to the coordinate system. The transformation can be any combination of translation, scaling, and rotation.

I mention *Graphics2D* here for completeness. I will not use any of the advanced capabilities of *Graphics2D* in this chapter, but I will cover a few of them in Section 13.2.

6.3.6 An Example

Let's use some of the material covered in this section to write a subclass of *JPanel* for use as a drawing surface. The panel can then be used in either an applet or a frame, as discussed in

Subsection 6.2.2. All the drawing will be done in the `paintComponent()` method of the panel class. The panel will draw multiple copies of a message on a black background. Each copy of the message is in a random color. Five different fonts are used, with different sizes and styles. The message can be specified in the constructor; if the default constructor is used, the message is the string "Java!". The panel works OK no matter what its size. Here is what the panel looks like:

There is one problem with the way this class works. When the panel's `paintComponent()` method is called, it chooses random colors, fonts, and locations for the messages. The information about which colors, fonts, and locations are used is not stored anywhere. The next time `paintComponent()` is called, it will make different random choices and will draw a different picture. A better approach would be to compute the contents of the picture elsewhere, outside the `paintComponent()` method. Information about the picture would be stored in instance variables, and the `paintComponent()` method would use that information to draw the picture. If `paintComponent()` is called twice, it should draw the same picture twice, unless the data has changed in the meantime. Unfortunately, to store the data for the picture in this applet, we would need to use either arrays, which will not be covered until Chapter 7, or off-screen images, which will not be covered until Chapter 13. Other examples in this chapter will suffer from the same problem.

The source for the panel class is shown below. I use an instance variable called `message` to hold the message that the panel will display. There are five instance variables of type *Font* that represent different sizes and styles of text. These variables are initialized in the constructor and are used in the `paintComponent()` method.

The `paintComponent()` method for the panel simply draws 25 copies of the message. For each copy, it chooses one of the five fonts at random, and it calls `g.setFont()` to select that font for drawing the text. It creates a random HSB color and uses `g.setColor()` to select that color for drawing. It then chooses random `(x,y)` coordinates for the location of the message. The `x` coordinate gives the horizontal position of the left end of the string. The formula used for the `x` coordinate, "`-50 + (int)(Math.random() * (width+40))`" gives a random integer in the range from `-50` to `width-10`. This makes it possible for the string to extend beyond the left edge or the right edge of the panel. Similarly, the formula for `y` allows the string to extend beyond the top and bottom of the applet.

Here is the complete source code for the `RandomStringsPanel`:

```
import java.awt.Color;
import java.awt.Font;
import java.awt.Graphics;
import javax.swing.JPanel;
```

```
/**
 * This panel displays 25 copies of a message.  The color and
 * position of each message is selected at random.  The font
 * of each message is randomly chosen from among five possible
 * fonts.  The messages are displayed on a black background.
 * <p>Note:  The style of drawing used here is poor, because every
 * time the paintComponent() method is called, new random values are
 * used.  This means that a different picture will be drawn each time.
 * <p>This panel is meant to be used as the content pane in
 * either an applet or a frame.
 */
public class RandomStringsPanel extends JPanel {

   private String message;  // The message to be displayed.  This can be set in
                            // the constructor.  If no value is provided in the
                            // constructor, then the string "Java!" is used.

   private Font font1, font2, font3, font4, font5;  // The five fonts.

   /**
    * Default constructor creates a panel that displays the message "Java!".
    */
   public RandomStringsPanel() {
      this(null);  // Call the other constructor, with parameter null.
   }

   /**
    * Constructor creates a panel to display 25 copies of a specified message.
    * @param messageString The message to be displayed.  If this is null,
    * then the default message "Java!" is displayed.
    */
   public RandomStringsPanel(String messageString) {

      message = messageString;
      if (message == null)
          message = "Java!";

      font1 = new Font("Serif", Font.BOLD, 14);
      font2 = new Font("SansSerif", Font.BOLD + Font.ITALIC, 24);
      font3 = new Font("Monospaced", Font.PLAIN, 30);
      font4 = new Font("Dialog", Font.PLAIN, 36);
      font5 = new Font("Serif", Font.ITALIC, 48);

      setBackground(Color.BLACK);

   }

   /**
    * The paintComponent method is responsible for drawing the content of the panel.
    * It draws 25 copies of the message string, using a random color, font, and
    * position for each string.
    */
   public void paintComponent(Graphics g) {

      super.paintComponent(g);  // Call the paintComponent method from the
                                // superclass, JPanel.  This simply fills the
                                // entire panel with the background color, black.
```

```
            int width = getWidth();
            int height = getHeight();

            for (int i = 0; i < 25; i++) {

                // Draw one string.  First, set the font to be one of the five
                // available fonts, at random.

                int fontNum = (int)(5*Math.random()) + 1;
                switch (fontNum) {
                   case 1:
                      g.setFont(font1);
                      break;
                   case 2:
                      g.setFont(font2);
                      break;
                   case 3:
                      g.setFont(font3);
                      break;
                   case 4:
                      g.setFont(font4);
                      break;
                   case 5:
                      g.setFont(font5);
                      break;
                 } // end switch

                 // Set the color to a bright, saturated color, with random hue.

                 float hue = (float)Math.random();
                 g.setColor( Color.getHSBColor(hue, 1.0F, 1.0F) );

                 // Select the position of the string, at random.

                 int x,y;
                 x = -50 + (int)(Math.random()*(width+40));
                 y = (int)(Math.random()*(height+20));

                 // Draw the message.

                 g.drawString(message,x,y);

            } // end for

         } // end paintComponent()

      }  // end class RandomStringsPanel
```

This class defines a panel, which is not something that can stand on its own. To see it on the screen, we have to use it in an applet or a frame. Here is a simple applet class that uses a *RandomStringsPanel* as its content pane:

```
import javax.swing.JApplet;

/**
 * A RandomStringsApplet displays 25 copies of a string, using random colors,
 * fonts, and positions for the copies.  The message can be specified as the
 * value of an applet param with name "message."  If no param with name
 * "message" is present, then the default message "Java!" is displayed.
```

```
 * The actual content of the applet is an object of type RandomStringsPanel.
 */
public class RandomStringsApplet extends JApplet {

   public void init() {
      String message = getParameter("message");  // If null, "Java" is used.
      RandomStringsPanel content = new RandomStringsPanel(message);
      setContentPane(content);
   }

}
```

Note that the message to be displayed in the applet can be set using an applet parameter when the applet is added to an HTML document. Using applets on Web pages was discussed in Subsection 6.2.4. Remember that to use the applet on a Web page, you must include both the panel class file, `RandomStringsPanel.class`, and the applet class file, `RandomStringsApplet.class`, in the same directory as the HTML document (or, alternatively, bundle the two class files into a jar file, and put the jar file in the document directory).

Instead of writing an applet, of course, we could use the panel in the window of a stand-alone application. You can find the source code for a main program that does this in the file *RandomStringsApp.java*.

6.4 Mouse Events

EVENTS ARE CENTRAL to programming for a graphical user interface. A GUI program doesn't have a `main()` routine that outlines what will happen when the program is run, in a step-by-step process from beginning to end. Instead, the program must be prepared to respond to various kinds of events that can happen at unpredictable times and in an order that the program doesn't control. The most basic kinds of events are generated by the mouse and keyboard. The user can press any key on the keyboard, move the mouse, or press a button on the mouse. The user can do any of these things at any time, and the computer has to respond appropriately.

In Java, events are represented by objects. When an event occurs, the system collects all the information relevant to the event and constructs an object to contain that information. Different types of events are represented by objects belonging to different classes. For example, when the user presses one of the buttons on a mouse, an object belonging to a class called *MouseEvent* is constructed. The object contains information such as the source of the event (that is, the component on which the user clicked), the `(x,y)` coordinates of the point in the component where the click occurred, the exact time of the click, and which button on the mouse was pressed. When the user presses a key on the keyboard, a *KeyEvent* is created. After the event object is constructed, it is passed as a parameter to a designated method. By writing that method, the programmer says what should happen when the event occurs.

As a Java programmer, you get a fairly high-level view of events. There is a lot of processing that goes on between the time that the user presses a key or moves the mouse and the time that a subroutine in your program is called to respond to the event. Fortunately, you don't need to know much about that processing. But you should understand this much: Even though your GUI program doesn't have a `main()` routine, there is a sort of main routine running somewhere that executes a loop of the form

```
while the program is still running:
    Wait for the next event to occur
    Call a subroutine to handle the event
```

This loop is called an ***event loop***. Every GUI program has an event loop. In Java, you don't have to write the loop. It's part of "the system." If you write a GUI program in some other language, you might have to provide a main routine that runs the event loop.

In this section, we'll look at handling mouse events in Java, and we'll cover the framework for handling events in general. The next section will cover keyboard-related events and timer events. Java also has other types of events, which are produced by GUI components. These will be introduced in Section 6.6.

6.4.1 Event Handling

For an event to have any effect, a program must detect the event and react to it. In order to detect an event, the program must "listen" for it. Listening for events is something that is done by an object called an ***event listener***. An event listener object must contain instance methods for handling the events for which it listens. For example, if an object is to serve as a listener for events of type *MouseEvent*, then it must contain the following method (among several others):

```
public void mousePressed(MouseEvent evt) { . . . }
```

The body of the method defines how the object responds when it is notified that a mouse button has been pressed. The parameter, `evt`, contains information about the event. This information can be used by the listener object to determine its response.

The methods that are required in a mouse event listener are specified in an `interface` named *MouseListener*. To be used as a listener for mouse events, an object must implement this *MouseListener* interface. Java `interfaces` were covered in Subsection 5.7.1. (To review briefly: An `interface` in Java is just a list of instance methods. A class can "implement" an interface by doing two things. First, the class must be declared to implement the interface, as in "`class MouseHandler implements MouseListener`" or "`class MyApplet extends JApplet implements MouseListener`". Second, the class must include a definition for each instance method specified in the interface. An `interface` can be used as the type for a variable or formal parameter. We say that an object implements the *MouseListener* interface if it belongs to a class that implements the *MouseListener* interface. Note that it is not enough for the object to include the specified methods. It must also belong to a class that is specifically declared to implement the interface.)

Many events in Java are associated with GUI components. For example, when the user presses a button on the mouse, the associated component is the one that the user clicked on. Before a listener object can "hear" events associated with a given component, the listener object must be registered with the component. If a *MouseListener* object, `mListener`, needs to hear mouse events associated with a *Component* object, `comp`, the listener must be ***registered*** with the component by calling

```
comp.addMouseListener(mListener);
```

The `addMouseListener()` method is an instance method in class *Component*, and so can be used with any GUI component object. In our first few examples, we will listen for events on a JPanel that is being used as a drawing surface.

The event classes, such as *MouseEvent*, and the listener interfaces, such as *MouseListener*, are defined in the package `java.awt.event`. This means that if you want to work with events, you should either include the line "`import java.awt.event.*;`" at the beginning of your source code file or import the individual classes and interfaces.

Admittedly, there is a large number of details to tend to when you want to use events. To summarize, you must

1. Put the import specification "`import java.awt.event.*;`" (or individual imports) at the beginning of your source code;
2. Declare that some class implements the appropriate listener interface, such as *MouseListener*;
3. Provide definitions in that class for the methods specified by the interface;
4. Register the listener object with the component that will generate the events by calling a method such as `addMouseListener()` in the component.

Any object can act as an event listener, provided that it implements the appropriate interface. A component can listen for the events that it itself generates. A panel can listen for events from components that are contained in the panel. A special class can be created just for the purpose of defining a listening object. Many people consider it to be good form to use anonymous inner classes to define listening objects (see Subsection 5.7.3). You will see all of these patterns in examples in this textbook.

6.4.2 MouseEvent and MouseListener

The *MouseListener* interface specifies five different instance methods:

```
public void mousePressed(MouseEvent evt);
public void mouseReleased(MouseEvent evt);
public void mouseClicked(MouseEvent evt);
public void mouseEntered(MouseEvent evt);
public void mouseExited(MouseEvent evt);
```

The `mousePressed` method is called as soon as the user presses down on one of the mouse buttons, and `mouseReleased` is called when the user releases a button. These are the two methods that are most commonly used, but any mouse listener object must define all five methods; you can leave the body of a method empty if you don't want to define a response. The `mouseClicked` method is called if the user presses a mouse button and then releases it, without moving the mouse. (When the user does this, all three routines—`mousePressed`, `mouseReleased`, and `mouseClicked`—will be called in that order.) In most cases, you should define `mousePressed` instead of `mouseClicked`. The `mouseEntered` and `mouseExited` methods are called when the mouse cursor enters or leaves the component. For example, if you want the component to change appearance whenever the user moves the mouse over the component, you could define these two methods.

As a first example, we will look at a small addition to the *RandomStringsPanel* example from the previous section. In the new version, the panel will repaint itself when the user clicks on it. In order for this to happen, a mouse listener should listen for mouse events on the panel, and when the listener detects a `mousePressed` event, it should respond by calling the `repaint()` method of the panel.

For the new version of the program, we need an object that implements the *MouseListener* interface. One way to create the object is to define a separate class, such as:

```
import java.awt.Component;
import java.awt.event.*;

/**
 * An object of type RepaintOnClick is a MouseListener that
 * will respond to a mousePressed event by calling the repaint()
 * method of the source of the event.  That is, a RepaintOnClick
```

```
 * object can be added as a mouse listener to any Component;
 * when the user clicks that component, the component will be
 * repainted.
 */
public class RepaintOnClick implements MouseListener {

   public void mousePressed(MouseEvent evt) {
      Component source = (Component)evt.getSource();
      source.repaint();  // Call repaint() on the Component that was clicked.
   }

   public void mouseClicked(MouseEvent evt) { }
   public void mouseReleased(MouseEvent evt) { }
   public void mouseEntered(MouseEvent evt) { }
   public void mouseExited(MouseEvent evt) { }

}
```

This class does three of the four things that we need to do in order to handle mouse events: First, it imports `java.awt.event.*` for easy access to event-related classes. Second, it is declared that the class "`implements MouseListener`". And third, it provides definitions for the five methods that are specified in the *MouseListener* interface. (Note that four of the five event-handling methods have empty definitions. We really only want to define a response to `mousePressed` events, but in order to implement the *MouseListener* interface, a class **must** define all five methods.)

We must do one more thing to set up the event handling for this example: We must register an event-handling object as a listener with the component that will generate the events. In this case, the mouse events that we are interested in will be generated by an object of type *RandomStringsPanel*. If `panel` is a variable that refers to the panel object, we can create a mouse listener object and register it with the panel with the statements:

```
RepaintOnClick listener = new RepaintOnClick();  // Create MouseListener object.
panel.addMouseListener(listener);  // Register MouseListener with the panel.
```

Once this is done, the `listener` object will be notified of mouse events on the panel. When a `mousePressed` event occurs, the `mousePressed()` method in the `listener` will be called. The code in this method calls the `repaint()` method in the component that is the source of the event, that is, in the panel. The result is that the `RandomStringsPanel` is repainted with its strings in new random colors, fonts, and positions.

Although we have written the *RepaintOnClick* class for use with our *RandomStringsPanel* example, the event-handling class contains no reference at all to the *RandomStringsPanel* class. How can this be? The `mousePressed()` method in class *RepaintOnClick* looks at the source of the event, and calls its `repaint()` method. If we have registered the *RepaintOnClick* object as a listener on a *RandomStringsPanel*, then it is that panel that is repainted. But the listener object could be used with any type of component, and it would work in the same way.

Similarly, the *RandomStringsPanel* class contains no reference to the *RepaintOnClick* class—in fact, *RandomStringsPanel* was written before we even knew anything about mouse events! The panel will send mouse events to any object that has registered with it as a mouse listener. It does not need to know anything about that object except that it is capable of receiving mouse events.

The relationship between an object that generates an event and an object that responds to that event is rather loose. The relationship is set up by registering one object to listen for

events from the other object. This is something that can potentially be done from outside both objects. Each object can be developed independently, with no knowledge of the internal operation of the other object. This is the essence of ***modular design***: Build a complex system out of modules that interact only in straightforward, easy to understand ways. Then each module is a separate design problem that can be tackled independently. Java's event-handling framework is designed to offer strong support for modular design.

To make this clearer, consider the application version of the `ClickableRandomStrings` program. I have included *RepaintOnClick* as a nested class, although it could just as easily be a separate class. The main point is that this program uses the same *RandomStringsPanel* class that was used in the original program, which did not respond to mouse clicks. The mouse handling has been "bolted on" to an existing class, without having to make any changes at all to that class:

```
import java.awt.Component;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import javax.swing.JFrame;

/**
 * Displays a window that shows 25 copies of the string "Java!" in
 * random colors, fonts, and positions.  The content of the window
 * is an object of type RandomStringsPanel.  When the user clicks
 * the window, the content of the window is repainted, with the
 * strings in newly selected random colors, fonts, and positions.
 */
public class ClickableRandomStringsApp {

   public static void main(String[] args) {
      JFrame window = new JFrame("Random Strings");
      RandomStringsPanel content = new RandomStringsPanel();
      content.addMouseListener( new RepaintOnClick() );  // Register mouse listener.
      window.setContentPane(content);
      window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      window.setLocation(100,75);
      window.setSize(300,240);
      window.setVisible(true);
   }

   private static class RepaintOnClick implements MouseListener {

      public void mousePressed(MouseEvent evt) {
         Component source = (Component)evt.getSource();
         source.repaint();
      }

      public void mouseClicked(MouseEvent evt) { }
      public void mouseReleased(MouseEvent evt) { }
      public void mouseEntered(MouseEvent evt) { }
      public void mouseExited(MouseEvent evt) { }

   }
}
```

6.4.3 Mouse Coordinates

Often, when a mouse event occurs, you want to know the location of the mouse cursor. This information is available from the *MouseEvent* parameter to the event-handling method, which contains instance methods that return information about the event. If `evt` is the parameter, then you can find out the coordinates of the mouse cursor by calling `evt.getX()` and `evt.getY()`. These methods return integers which give the `x` and `y` coordinates where the mouse cursor was positioned at the time when the event occurred. The coordinates are expressed in the coordinate system of the component that generated the event, where the top left corner of the component is (0,0).

The user can hold down certain ***modifier keys*** while using the mouse. The possible modifier keys include: the Shift key, the Control key, the Alt key (called the Option key on the Mac), and the Meta key (called the Command or Apple key on the Mac). You might want to respond to a mouse event differently when the user is holding down a modifier key. The boolean-valued instance methods `evt.isShiftDown()`, `evt.isControlDown()`, `evt.isAltDown()`, and `evt.isMetaDown()` can be called to test whether the modifier keys are pressed.

You might also want to have different responses depending on whether the user presses the left mouse button, the middle mouse button, or the right mouse button. Now, not every mouse has a middle button and a right button, so Java handles the information in a peculiar way. It treats pressing the right button as equivalent to holding down the Meta key while pressing the left mouse button. That is, if the right button is pressed, then the instance method `evt.isMetaDown()` will return `true` (even if the Meta key is not pressed). Similarly, pressing the middle mouse button is equivalent to holding down the Alt key. In practice, what this really means is that pressing the right mouse button under Windows or Linux is equivalent to holding down the Command key while pressing the mouse button on the Mac. A program tests for either of these by calling `evt.isMetaDown()`.

As an example, consider a *JPanel* that does the following: Clicking on the panel with the left mouse button will place a red rectangle on the panel at the point where the mouse was clicked. Clicking with the right mouse button (or holding down the Command key while clicking on a Mac) will place a blue oval on the applet. Holding down the Shift key while clicking will clear the panel by removing all the shapes that have been placed.

There are several ways to write this example. There could be a separate class to handle mouse events, as in the previous example. However, in this case, I decided to let the panel itself respond to mouse events. Any object can be a mouse listener, as long as it implements the *MouseListener* interface. In this case, the panel class implements the *MouseListener* interface, so the object that represents the main panel of the program can be the mouse listener for the program. The constructor for the panel class contains the statement

```
addMouseListener(this);
```

which is equivalent to saying `this.addMouseListener(this)`. Now, the ordinary way to register a mouse listener is to say `X.addMouseListener(Y)` where `Y` is the listener and `X` is the component that will generate the mouse events. In the statement `addMouseListener(this)`, both roles are played by `this`; that is, "this object" (the panel) is generating mouse events and is also listening for those events. Although this might seem a little strange, you should get used to seeing things like this. In a large program, however, it's usually a better idea to write a separate class to do the listening in order to have a more organized division of responsibilities.

The source code for the panel class is shown below. You should check how the instance methods in the *MouseEvent* object are used. You can also check for the Four Steps of Event

Handling ("`import java.awt.event.*`", "`implements MouseListener`", definitions for the event-handling methods, and "`addMouseListener`"):

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * A simple demonstration of MouseEvents.  Shapes are drawn
 * on a black background when the user clicks the panel.  If
 * the user Shift-clicks, the applet is cleared.  If the user
 * right-clicks the applet, a blue oval is drawn.  Otherwise,
 * when the user clicks, a red rectangle is drawn.  The contents of
 * the panel are not persistent.  For example, they might disappear
 * if the panel is resized or is covered and uncovered.
 */
public class SimpleStamperPanel extends JPanel implements MouseListener {

   /**
    * This constructor simply sets the background color of the panel to be black
    * and sets the panel to listen for mouse events on itself.
    */
   public SimpleStamperPanel() {
      setBackground(Color.BLACK);
      addMouseListener(this);
   }


   /**
    *  Since this panel has been set to listen for mouse events on itself,
    *  this method will be called when the user clicks the mouse on the panel.
    *  This method is part of the MouseListener interface.
    */
   public void mousePressed(MouseEvent evt) {

      if ( evt.isShiftDown() ) {
            // The user was holding down the Shift key.  Just repaint the panel.
            // Since this class does not define a paintComponent() method, the
            // method from the superclass, JPanel, is called.  That method simply
            // fills the panel with its background color, which is black.  The
            // effect is to clear the panel.
         repaint();
         return;
      }

      int x = evt.getX();  // x-coordinate where user clicked.
      int y = evt.getY();  // y-coordinate where user clicked.

      Graphics g = getGraphics();  // Graphics context for drawing directly.
                                   // NOTE:  This is considered to be bad style!

      if ( evt.isMetaDown() ) {
             // User right-clicked at the point (x,y). Draw a blue oval centered
             // at the point (x,y). (A black outline around the oval will make it
             // more distinct when shapes overlap.)
         g.setColor(Color.BLUE);  // Blue interior.
         g.fillOval( x - 30, y - 15, 60, 30 );
```

```
            g.setColor(Color.BLACK); // Black outline.
            g.drawOval( x - 30, y - 15, 60, 30 );
         }
         else {
               // User left-clicked (or middle-clicked) at (x,y).
               // Draw a red rectangle centered at (x,y).
            g.setColor(Color.RED);   // Red interior.
            g.fillRect( x - 30, y - 15, 60, 30 );
            g.setColor(Color.BLACK); // Black outline.
            g.drawRect( x - 30, y - 15, 60, 30 );
         }

         g.dispose();  // We are finished with the graphics context, so dispose of it.

      } // end mousePressed();


      // The next four empty routines are required by the MouseListener interface.
      // They don't do anything in this class, so their definitions are empty.

      public void mouseEntered(MouseEvent evt) { }
      public void mouseExited(MouseEvent evt) { }
      public void mouseClicked(MouseEvent evt) { }
      public void mouseReleased(MouseEvent evt) { }

   } // end class SimpleStamperPanel
```

Note, by the way, that this class violates the rule that all drawing should be done in a `paintComponent()` method. The rectangles and ovals are drawn directly in the `mousePressed()` routine. To make this possible, I need to obtain a graphics context by saying "`g = getGraphics()`". After using `g` for drawing, I call `g.dispose()` to inform the operating system that I will no longer be using `g` for drawing. It is a good idea to do this to free the system resources that are used by the graphics context. I do not advise doing this type of direct drawing if it can be avoided, but you can see that it does work in this case, and at this point we really have no other way to write this example.

6.4.4 MouseMotionListeners and Dragging

Whenever the mouse is moved, it generates events. The operating system of the computer detects these events and uses them to move the mouse cursor on the screen. It is also possible for a program to listen for these "mouse motion" events and respond to them. The most common reason to do so is to implement ***dragging***. Dragging occurs when the user moves the mouse while holding down a mouse button.

The methods for responding to mouse motion events are defined in an interface named *MouseMotionListener*. This interface specifies two event-handling methods:

```
public void mouseDragged(MouseEvent evt);
public void mouseMoved(MouseEvent evt);
```

The `mouseDragged` method is called if the mouse is moved while a button on the mouse is pressed. If the mouse is moved while no mouse button is down, then `mouseMoved` is called instead. The parameter, `evt`, is an object of type *MouseEvent*. It contains the `x` and `y` coordinates of the mouse's location. As long as the user continues to move the mouse, one of these methods will be called over and over. (So many events are generated that it would be inefficient for a program to hear them all, if it doesn't want to do anything in response. This is why the

mouse motion event-handlers are defined in a separate interface from the other mouse events: You can listen for the mouse events defined in *MouseListener* without automatically hearing all mouse motion events as well.)

If you want your program to respond to mouse motion events, you must create an object that implements the *MouseMotionListener* interface, and you must register that object to listen for events. The registration is done by calling a component's `addMouseMotionListener()` method. The object will then listen for `mouseDragged` and `mouseMoved` events associated with that component. In most cases, the listener object will also implement the *MouseListener* interface so that it can respond to the other mouse events as well.

To get a better idea of how mouse events work, you should try the *SimpleTrackMouseApplet* in the on-line version of this section. The applet is programmed to respond to any of the seven different kinds of mouse events by displaying the coordinates of the mouse, the type of event, and a list of the modifier keys that are down (Shift, Control, Meta, and Alt). You can experiment with the applet to see what happens when you use the mouse on the applet. (Alternatively, you could run the stand-alone application version of the program, *SimpleTrackMouse.java*.)

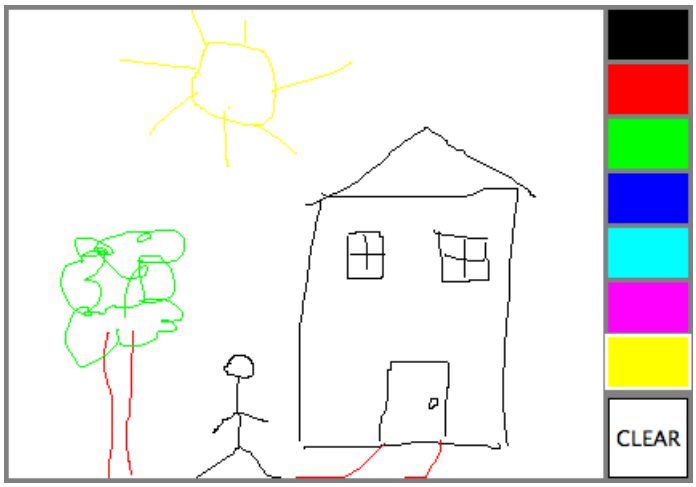
The source code for the program can be found in *SimpleTrackMousePanel.java*, which defines the panel that is used as the content pane, and in *SimpleTrackMouseApplet.java*, which defines the applet class. The panel class includes a nested class, *MouseHandler*, that defines the mouse-handling object. I encourage you to read the source code. You should now be familiar with all the techniques that it uses.

It is interesting to look at what a program needs to do in order to respond to dragging operations. In general, the response involves three methods: `mousePressed()`, `mouseDragged()`, and `mouseReleased()`. The dragging gesture starts when the user presses a mouse button, it continues while the mouse is dragged, and it ends when the user releases the button. This means that the programming for the response to one dragging gesture must be spread out over the three methods! Furthermore, the `mouseDragged()` method can be called many times as the mouse moves. To keep track of what is going on between one method call and the next, you need to set up some instance variables. In many applications, for example, in order to process a `mouseDragged` event, you need to remember the previous coordinates of the mouse. You can store this information in two instance variables `prevX` and `prevY` of type **int**. It can also be useful to save the starting coordinates, where the original `mousePressed` event occurred, in instance variables. I also suggest having a **boolean** variable, `dragging`, which is set to `true` while a dragging gesture is being processed. This is necessary because in many applications, not every `mousePressed` event starts a dragging operation to which you want to respond. The `mouseDragged` and `mouseReleased` methods can use the value of `dragging` to check whether a drag operation is actually in progress. You might need other instance variables as well, but in general outline, a class that handles mouse dragging looks like this:

```
import java.awt.event.*;

public class MouseDragHandler implements MouseListener, MouseMotionListener {

   private int startX, startY; // Point where the original mousePress occurred.
   private int prevX, prevY;   // Most recently processed mouse coords.
   private boolean dragging;   // Set to true when dragging is in process.
   . . . // other instance variables for use in dragging

   public void mousePressed(MouseEvent evt) {
      if ( we-want-to-start-dragging ) {
          dragging = true;
          startX = evt.getX();  // Remember starting position.
```

```
            startY = evt.getY();
            prevX = startX;       // Remember most recent coords.
            prevY = startY;
               .
               . // Other processing.
               .
        }
    }

    public void mouseDragged(MouseEvent evt) {
        if ( dragging == false )  // First, check if we are
            return;               //   processing a dragging gesture.
        int x = evt.getX(); // Current position of Mouse.
        int y = evt.getY();
          .
          .  // Process a mouse movement from (prevX, prevY) to (x,y).
          .
        prevX = x;  // Remember the current position for the next call.
        prevY = y;
    }

    public void mouseReleased(MouseEvent evt) {
        if ( dragging == false )  // First, check if we are
            return;               //   processing a dragging gesture.
        dragging = false;  // We are done dragging.
         .
         .  // Other processing and clean-up.
         .
    }

}
```

As an example, let's look at a typical use of dragging: allowing the user to sketch a curve by dragging the mouse. This example also shows many other features of graphics and mouse processing. In the program, you can draw a curve by dragging the mouse on a large white drawing area, and you can select a color for drawing by clicking on one of several colored rectangles to the right of the drawing area. The complete source code can be found in *SimplePaint.java*, which can be run as a stand-alone application, and you can find an applet version in the on-line version of this section. Here is a picture of the program:



I will discuss a few aspects of the source code here, but I encourage you to read it carefully in its entirety. There are lots of informative comments in the source code. (The source code uses one unusual technique: It defines a subclass of *JApplet*, but it also includes a `main()` routine. The `main()` routine has nothing to do with the class's use as an applet, but it makes it possible to run the class as a stand-alone application. When this is done, the application opens a window that shows the same panel that would be shown in the applet version. This example thus shows how to write a single file that can be used either as a stand-alone application or as an applet.)

The panel class for this example is designed to work for any reasonable size, that is, unless the panel is too small. This means that coordinates are computed in terms of the actual width and height of the panel. (The width and height are obtained by calling `getWidth()` and `getHeight()`.) This makes things quite a bit harder than they would be if we assumed some particular fixed size for the panel. Let's look at some of these computations in detail. For example, the large white drawing area extends from `y = 3` to `y = height - 3` vertically and from `x = 3` to `x = width - 56` horizontally. These numbers are needed in order to interpret the meaning of a mouse click. They take into account a gray border around the panel and the color palette along the right edge of the panel. The gray border is 3 pixels wide. The colored rectangles are 50 pixels wide. Together with the 3-pixel border around the panel and a 3-pixel divider between the drawing area and the colored rectangles, this adds up to put the right edge of the drawing area 56 pixels from the right edge of the panel.

A white square labeled "`CLEAR`" occupies a 50-by-50 pixel region beneath the colored rectangles on the right edge of the panel. Allowing for this square, we can figure out how much vertical space is available for the seven colored rectangles, and then divide that space by 7 to get the vertical space available for each rectangle. This quantity is represented by a variable, `colorSpace`. Out of this space, 3 pixels are used as spacing between the rectangles, so the height of each rectangle is `colorSpace - 3`. The top of the `N`-th rectangle is located `(N*colorSpace + 3)` pixels down from the top of the panel, assuming that we count the rectangles starting with zero. This is because there are `N` rectangles above the `N`-th rectangle, each of which uses `colorSpace` pixels. The extra 3 is for the border at the top of the panel. After all that, we can write down the command for drawing the `N`-th rectangle:

```
g.fillRect(width - 53, N*colorSpace + 3, 50, colorSpace - 3);
```

That was not easy! But it shows the kind of careful thinking and precision graphics that are sometimes necessary to get good results.

The mouse in this program is used to do three different things: Select a color, clear the drawing, and draw a curve. Only the third of these involves dragging, so not every mouse click will start a dragging operation. The `mousePressed()` method has to look at the `(x,y)` coordinates where the mouse was clicked and decide how to respond. If the user clicked on the `CLEAR` rectangle, the drawing area is cleared by calling `repaint()`. If the user clicked somewhere in the strip of colored rectangles, the corresponding color is selected for drawing. This involves computing which color the user clicked on, which is done by dividing the `y` coordinate by `colorSpace`. Finally, if the user clicked on the drawing area, a drag operation is initiated. In this case, a boolean variable, `dragging`, is set to `true` so that the `mouseDragged` and `mouseReleased` methods will know that a curve is being drawn. The code for this follows the general form given above. The actual drawing of the curve is done in the `mouseDragged()` method, which draws a line from the previous location of the mouse to its current location. Some effort is required to make sure that the line does not extend beyond the white drawing area of the panel. This is not automatic, since as far as the computer is concerned, the border and

the color bar are part of the drawing surface. If the user drags the mouse outside the drawing area while drawing a line, the `mouseDragged()` routine changes the `x` and `y` coordinates to make them lie within the drawing area.

6.4.5 Anonymous Event Handlers and Adapter Classes

As I mentioned above, it is a fairly common practice to use anonymous inner classes to define listener objects. As discussed in Subsection 5.7.3, a special form of the `new` operator is used to create an object that belongs to an anonymous class. For example, a mouse listener object can be created with an expression of the form:

```
new MouseListener() {
   public void mousePressed(MouseEvent evt) { . . . }
   public void mouseReleased(MouseEvent evt) { . . . }
   public void mouseClicked(MouseEvent evt) { . . . }
   public void mouseEntered(MouseEvent evt) { . . . }
   public void mouseExited(MouseEvent evt) { . . . }
}
```

This is all just one long expression that both defines an unnamed class and creates an object that belongs to that class. To use the object as a mouse listener, it can be passed as the parameter to some component's `addMouseListener()` method in a command of the form:

```
component.addMouseListener( new MouseListener() {
      public void mousePressed(MouseEvent evt) { . . . }
      public void mouseReleased(MouseEvent evt) { . . . }
      public void mouseClicked(MouseEvent evt) { . . . }
      public void mouseEntered(MouseEvent evt) { . . . }
      public void mouseExited(MouseEvent evt) { . . . }
   } );
```

Now, in a typical application, most of the method definitions in this class will be empty. A class that implements an `interface` must provide definitions for all the methods in that interface, even if the definitions are empty. To avoid the tedium of writing empty method definitions in cases like this, Java provides ***adapter classes***. An adapter class implements a listener interface by providing empty definitions for all the methods in the interface. An adapter class is useful only as a basis for making subclasses. In the subclass, you can define just those methods that you actually want to use. For the remaining methods, the empty definitions that are provided by the adapter class will be used. The adapter class for the *MouseListener* interface is named `MouseAdapter`. For example, if you want a mouse listener that only responds to mouse-pressed events, you can use a command of the form:

```
component.addMouseListener( new MouseAdapter() {
      public void mousePressed(MouseEvent evt) { . . . }
   } );
```

To see how this works in a real example, let's write another version of the *ClickableRandomStringsApp* application from Subsection 6.4.2. This version uses an anonymous class based on *MouseAdapter* to handle mouse events:

```
import java.awt.Component;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
```

```
import javax.swing.JFrame;

public class ClickableRandomStringsApp {

   public static void main(String[] args) {
      JFrame window = new JFrame("Random Strings");
      RandomStringsPanel content = new RandomStringsPanel();

      content.addMouseListener( new MouseAdapter() {
            // Register a mouse listener that is defined by an anonymous subclass
            // of MouseAdapter.  This replaces the RepaintOnClick class that was
            // used in the original version.
         public void mousePressed(MouseEvent evt) {
            Component source = (Component)evt.getSource();
            source.repaint();
         }
      } );

      window.setContentPane(content);
      window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      window.setLocation(100,75);
      window.setSize(300,240);
      window.setVisible(true);
   }

}
```

There is also an adapter class for mouse motion listeners, *MouseMotionAdapter*, which implements *MouseMotionListener* and defines empty versions of `mouseDragged()` and `mouseMoved()`. In Java 6 and later, the *MouseAdapter* class actually implements *MouseMotionListener* as well as *MouseListener*, so there is less use for *MouseMotionAdapter*.

Anonymous inner classes can be used for other purposes besides event handling. For example, suppose that you want to define a subclass of *JPanel* to represent a drawing surface. The subclass will only be used once. It will redefine the `paintComponent()` method, but will make no other changes to *JPanel*. It might make sense to define the subclass as an anonymous inner class. As an example, I present *HelloWorldGUI4.java*. This version is a variation of *HelloWorldGUI2.java* that uses anonymous inner classes where the original program uses ordinary, named nested classes:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * A simple GUI program that creates and opens a JFrame containing
 * the message "Hello World" and an "OK" button.  When the user clicks
 * the OK button, the program ends.  This version uses anonymous
 * classes to define the message display panel and the action listener
 * object.  Compare to HelloWorldGUI2, which uses nested classes.
 */
public class HelloWorldGUI4 {

   /**
    * The main program creates a window containing a display panel
    * and a button that will end the program when the user clicks it.
    */
```

```
    public static void main(String[] args) {

       JPanel displayPanel = new JPanel() {
              // An anonymous subclass of JPanel that displays "Hello World!".
          public void paintComponent(Graphics g) {
             super.paintComponent(g);
             g.drawString( "Hello World!", 20, 30 );
          }
       };

       JButton okButton = new JButton("OK");

       okButton.addActionListener( new ActionListener() {
              // An anonymous class that defines the listener object.
          public void actionPerformed(ActionEvent e) {
             System.exit(0);
          }
       } );

       JPanel content = new JPanel();
       content.setLayout(new BorderLayout());
       content.add(displayPanel, BorderLayout.CENTER);
       content.add(okButton, BorderLayout.SOUTH);

       JFrame window = new JFrame("GUI Test");
       window.setContentPane(content);
       window.setSize(250,100);
       window.setLocation(100,100);
       window.setVisible(true);

    }

}
```

6.5 Timers, KeyEvents, and State Machines

Not every event is generated by an action on the part of the user. Events can also be generated by objects as part of their regular programming, and these events can be monitored by other objects so that they can take appropriate actions when the events occur. One example of this is the class `javax.swing.Timer`. A *Timer* generates events at regular intervals. These events can be used to drive an animation or to perform some other task at regular intervals. We will begin this section with a look at timer events and animation. We will then look at another type of basic user-generated event: the *KeyEvents* that are generated when the user types on the keyboard. The example at the end of the section uses both a timer and keyboard events to implement a simple game and introduces the important idea of ***state machines***.

6.5.1 Timers and Animation

An object belonging to the class `javax.swing.Timer` exists only to generate events. A *Timer*, by default, generates a sequence of events with a fixed delay between each event and the next. (It is also possible to set a *Timer* to emit a single event after a specified time delay; in that case, the timer is being used as an "alarm.") Each event belongs to the class *ActionEvent*. An object

that is to listen for the events must implement the interface *ActionListener*, which defines just one method:

```
public void actionPerformed(ActionEvent evt)
```

To use a *Timer*, you must create an object that implements the *ActionListener* interface. That is, the object must belong to a class that is declared to "`implement ActionListener`", and that class must define the `actionPerformed` method. Then, if the object is set to listen for events from the timer, the code in the listener's `actionPerformed` method will be executed every time the timer generates an event.

Since there is no point to having a timer without having a listener to respond to its events, the action listener for a timer is specified as a parameter in the timer's constructor. The time delay between timer events is also specified in the constructor. If `timer` is a variable of type *Timer*, then the statement

```
timer = new Timer( millisDelay, listener );
```

creates a timer with a delay of `millisDelay` milliseconds between events (where 1000 milliseconds equal one second). Events from the timer are sent to the `listener`. (`millisDelay` must be of type **int**, and `listener` must be of type *ActionListener*.) Note that a timer is not guaranteed to deliver events at precisely regular intervals. If the computer is busy with some other task, an event might be delayed or even dropped altogether.

A timer does not automatically start generating events when the timer object is created. The `start()` method in the timer must be called to tell the timer to start generating events. The timer's `stop()` method can be used to turn the stream of events off—it can be restarted by calling `start()` again.

* * *

One application of timers is computer animation. A computer animation is just a sequence of still images, presented to the user one after the other. If the time between images is short, and if the change from one image to another is not too great, then the user perceives continuous motion. The easiest way to do animation in Java is to use a *Timer* to drive the animation. Each time the timer generates an event, the next frame of the animation is computed and drawn on the screen—the code that implements this goes in the `actionPerformed` method of an object that listens for events from the timer.

Our first example of using a timer is not exactly an animation, but it does display a new image for each timer event. The program shows randomly generated images that vaguely resemble works of abstract art. In fact, the program draws a new random image every time its `paintComponent()` method is called, and the response to a timer event is simply to call `repaint()`, which in turn triggers a call to `paintComponent`. The work of the program is done in a subclass of *JPanel*, which starts like this:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class RandomArtPanel extends JPanel {

   /**
    * A RepaintAction object calls the repaint method of this panel each
    * time its actionPerformed() method is called.  An object of this
    * type is used as an action listener for a Timer that generates an
    * ActionEvent every four seconds.  The result is that the panel is
```

```
 * redrawn every four seconds.
 */
private class RepaintAction implements ActionListener {
   public void actionPerformed(ActionEvent evt) {
      repaint();  // Call the repaint() method in the panel class.
   }
}

/**
 * The constructor creates a timer with a delay time of four seconds
 * (4000 milliseconds), and with a RepaintAction object as its
 * ActionListener.  It also starts the timer running.
 */
public RandomArtPanel() {
   RepaintAction action = new RepaintAction();
   Timer timer = new Timer(4000, action);
   timer.start();
}

/**
 * The paintComponent() method fills the panel with a random shade of
 * gray and then draws one of three types of random "art".  The type
 * of art to be drawn is chosen at random.
 */
public void paintComponent(Graphics g) {
    .
    .   // The rest of the class is omitted
    .
```

You can find the full source code for this class in the file *RandomArtPanel.java*; An application version of the program is *RandomArt.java*, while the applet version is *RandomArtApplet.java*. You can see the applet version in the on-line version of this section.

Later in this section, we will use a timer to drive the animation in a simple computer game.

6.5.2 Keyboard Events

In Java, user actions become events in a program. These events are associated with GUI components. When the user presses a button on the mouse, the event that is generated is associated with the component that contains the mouse cursor. What about keyboard events? When the user presses a key, what component is associated with the key event that is generated?

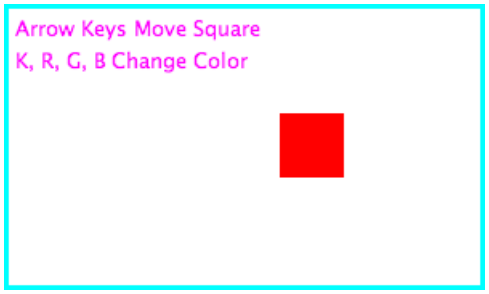
A GUI uses the idea of ***input focus*** to determine the component associated with keyboard events. At any given time, exactly one interface element on the screen has the input focus, and that is where all keyboard events are directed. If the interface element happens to be a Java component, then the information about the keyboard event becomes a Java object of type *KeyEvent*, and it is delivered to any listener objects that are listening for `KeyEvents` associated with that component. The necessity of managing input focus adds an extra twist to working with keyboard events.

It's a good idea to give the user some visual feedback about which component has the input focus. For example, if the component is the typing area of a word-processor, the feedback is usually in the form of a blinking text cursor. Another common visual clue is to draw a brightly colored border around the edge of a component when it has the input focus, as I do in the examples given later in this section.

A component that wants to have the input focus can call the method `requestFocus()`, which is defined in the `Component` class. Calling this method does not absolutely guarantee that the component will actually get the input focus. Several components might request the focus; only one will get it. This method should only be used in certain circumstances in any case, since it can be a rude surprise to the user to have the focus suddenly pulled away from a component that the user is working with. In a typical user interface, the user can choose to give the focus to a component by clicking on that component with the mouse. And pressing the tab key will often move the focus from one component to another.

Some components do not automatically request the input focus when the user clicks on them. To solve this problem, a program has to register a mouse listener with the component to detect user clicks. In response to a user click, the `mousePressed()` method should call `requestFocus()` for the component. This is true, in particular, for the components that are used as drawing surfaces in the examples in this chapter. These components are defined as subclasses of *JPanel*, and *JPanel* objects do not receive the input focus automatically. If you want to be able to use the keyboard to interact with a *JPanel* named `drawingSurface`, you have to register a listener to listen for mouse events on the `drawingSurface` and call `drawingSurface.requestFocus()` in the `mousePressed()` method of the listener object.

As our first example of processing key events, we look at a simple program in which the user moves a square up, down, left, and right by pressing arrow keys. When the user hits the 'R', 'G', 'B', or 'K' key, the color of the square is set to red, green, blue, or black, respectively. Of course, none of these key events are delivered to the panel unless it has the input focus. The panel in the program changes its appearance when it has the input focus: When it does, a cyan-colored border is drawn around the panel; when it does not, a gray-colored border is drawn. Also, the panel displays a different message in each case. If the panel does not have the input focus, the user can give the input focus to the panel by clicking on it. The complete source code for this example can be found in the file *KeyboardAndFocusDemo.java*. I will discuss some aspects of it below. After reading this section, you should be able to understand the source code in its entirety. Here is what the program looks like in its focused state:



In Java, keyboard event objects belong to a class called *KeyEvent*. An object that needs to listen for *KeyEvents* must implement the interface named *KeyListener*. Furthermore, the object must be registered with a component by calling the component's `addKeyListener()` method. The registration is done with the command "`component.addKeyListener(listener);`" where `listener` is the object that is to listen for key events, and `component` is the object that will generate the key events (when it has the input focus). It is possible for `component` and `listener` to be the same object. All this is, of course, directly analogous to what you learned about mouse events in the previous section. The *KeyListener* interface defines the following methods, which must be included in any class that implements *KeyListener*:

```
public void keyPressed(KeyEvent evt);
public void keyReleased(KeyEvent evt);
public void keyTyped(KeyEvent evt);
```

Java makes a careful distinction between *the keys that you press* and *the characters that you type*. There are lots of keys on a keyboard: letter keys, number keys, modifier keys such as Control and Shift, arrow keys, page up and page down keys, keypad keys, function keys, and so on. In many cases, pressing a key does not type a character. On the other hand, typing a character sometimes involves pressing several keys. For example, to type an uppercase 'A', you have to press the Shift key and then press the A key before releasing the Shift key. On my Mac OS computer, I can type an accented e, by holding down the Option key, pressing the E key, releasing the Option key, and pressing E again. Only one character was typed, but I had to perform three key-presses and I had to release a key at the right time. In Java, there are three types of *KeyEvent*. The types correspond to pressing a key, releasing a key, and typing a character. The `keyPressed` method is called when the user presses a key, the `keyReleased` method is called when the user releases a key, and the `keyTyped` method is called when the user types a character (whether that's done with one key press or several). Note that one user action, such as pressing the E key, can be responsible for two events, a `keyPressed` event and a `keyTyped` event. Typing an upper case 'A' can generate two `keyPressed` events, two `keyReleased` events, and one `keyTyped` event.

Usually, it is better to think in terms of two separate streams of events, one consisting of `keyPressed` and `keyReleased` events and the other consisting of `keyTyped` events. For some applications, you want to monitor the first stream; for other applications, you want to monitor the second one. Of course, the information in the `keyTyped` stream could be extracted from the `keyPressed/keyReleased` stream, but it would be difficult (and also system-dependent to some extent). Some user actions, such as pressing the Shift key, can only be detected as `keyPressed` events. I used to have a computer solitaire game that highlighted every card that could be moved, when I held down the Shift key. You can do something like that in Java by hiliting the cards when the Shift key is pressed and removing the highlight when the Shift key is released.

There is one more complication. Usually, when you hold down a key on the keyboard, that key will ***auto-repeat***. This means that it will generate multiple `keyPressed` events, as long as it is held down. It can also generate multiple `keyTyped` events. For the most part, this will not affect your programming, but you should not expect every `keyPressed` event to have a corresponding `keyReleased` event.

Every key on the keyboard has an integer code number. (Actually, this is only true for keys that Java knows about. Many keyboards have extra keys that can't be used with Java.) When the `keyPressed` or `keyReleased` method is called, the parameter, `evt`, contains the code of the key that was pressed or released. The code can be obtained by calling the function `evt.getKeyCode()`. Rather than asking you to memorize a table of code numbers, Java provides a named constant for each key. These constants are defined in the *KeyEvent* class. For example the constant for the shift key is `KeyEvent.VK_SHIFT`. If you want to test whether the key that the user pressed is the Shift key, you could say "`if (evt.getKeyCode() == KeyEvent.VK_SHIFT)`". The key codes for the four arrow keys are `KeyEvent.VK_LEFT`, `KeyEvent.VK_RIGHT`, `KeyEvent.VK_UP`, and `KeyEvent.VK_DOWN`. Other keys have similar codes. (The "VK" stands for "Virtual Keyboard". In reality, different keyboards use different key codes, but Java translates the actual codes from the keyboard into its own "virtual" codes. Your program only sees these virtual key codes, so it will work with various keyboards on

various platforms without modification.)

In the case of a `keyTyped` event, you want to know which character was typed. This information can be obtained from the parameter, `evt`, in the `keyTyped` method by calling the function `evt.getKeyChar()`. This function returns a value of type **char** representing the character that was typed.

In the `KeyboardAndFocusDemo` program, I use the `keyPressed` routine to respond when the user presses one of the arrow keys. The applet includes instance variables, `squareLeft` and `squareTop`, that give the position of the upper left corner of the movable square. When the user presses one of the arrow keys, the `keyPressed` routine modifies the appropriate instance variable and calls `repaint()` to redraw the panel with the square in its new position. Note that the values of `squareLeft` and `squareTop` are restricted so that the square never moves outside the white area of the panel:

```
/**
 * This is called each time the user presses a key while the panel has
 * the input focus.  If the key pressed was one of the arrow keys,
 * the square is moved (except that it is not allowed to move off the
 * edge of the panel, allowing for a 3-pixel border).
 */
public void keyPressed(KeyEvent evt) {

   int key = evt.getKeyCode();  // keyboard code for the pressed key

   if (key == KeyEvent.VK_LEFT) {  // left-arrow key; move the square left
      squareLeft -= 8;
      if (squareLeft < 3)
         squareLeft = 3;
      repaint();
   }
   else if (key == KeyEvent.VK_RIGHT) {  // right-arrow key; move the square right
      squareLeft += 8;
      if (squareLeft > getWidth() - 3 - SQUARE_SIZE)
         squareLeft = getWidth() - 3 - SQUARE_SIZE;
      repaint();
   }
   else if (key == KeyEvent.VK_UP) {  // up-arrow key; move the square up
      squareTop -= 8;
      if (squareTop < 3)
         squareTop = 3;
      repaint();
   }
   else if (key == KeyEvent.VK_DOWN) {  // down-arrow key; move the square down
      squareTop += 8;
      if (squareTop > getHeight() - 3 - SQUARE_SIZE)
         squareTop = getHeight() - 3 - SQUARE_SIZE;
      repaint();
   }

}  // end keyPressed()
```

Color changes—which happen when the user types the characters 'R', 'G', 'B', and 'K', or the lower case equivalents—are handled in the `keyTyped` method. I won't include it here, since it is so similar to the `keyPressed` method. Finally, to complete the *KeyListener* interface, the

`keyReleased` method must be defined. In the sample program, the body of this method is empty since the applet does nothing in response to `keyReleased` events.

6.5.3 Focus Events

If a component is to change its appearance when it has the input focus, it needs some way to know when it has the focus. In Java, objects are notified about changes of input focus by events of type *FocusEvent*. An object that wants to be notified of changes in focus can implement the *FocusListener* interface. This interface declares two methods:

```
public void focusGained(FocusEvent evt);
public void focusLost(FocusEvent evt);
```

Furthermore, the `addFocusListener()` method must be used to set up a listener for the focus events. When a component gets the input focus, it calls the `focusGained()` method of any object that has been registered with that component as a *FocusListener*. When it loses the focus, it calls the listener's `focusLost()` method. Sometimes, it is the component itself that listens for focus events.

In the sample `KeyboardAndFocusDemo` program, the response to a focus event is simply to redraw the panel. The `paintComponent()` method checks whether the panel has the input focus by calling the **boolean**-valued function `hasFocus()`, which is defined in the *Component* class, and it draws a different picture depending on whether or not the panel has the input focus. The net result is that the appearance of the panel changes when the panel gains or loses focus. The methods from the *FocusListener* interface are defined simply as:

```
public void focusGained(FocusEvent evt) {
       // The panel now has the input focus.
   repaint();  // will redraw with a new message and a cyan border
}

public void focusLost(FocusEvent evt) {
      // The panel has now lost the input focus.
   repaint();  // will redraw with a new message and a gray border
}
```

The other aspect of handling focus is to make sure that the `panel` gets the focus when the user clicks on it. To do this, the panel implements the `MouseListener` interface and listens for mouse events on itself. It defines a `mousePressed` routine that asks that the input focus be given to the panel:

```
public void mousePressed(MouseEvent evt) {
   requestFocus();
}
```

The other four methods of the `mouseListener` interface are defined to be empty. Note that the panel implements three different listener interfaces, *KeyListener*, *FocusListener*, and *MouseListener*, and the constructor in the panel class registers itself to listen for all three types of events with the statements:

```
addKeyListener(this);
addFocusListener(this);
addMouseListener(this);
```

There are, of course, other ways to organize this example. It would be possible, for example, to use a nested class to define the listening object. Or anonymous classes could be used to define separate listening objects for each type of event. In my next example, I will take the latter approach.

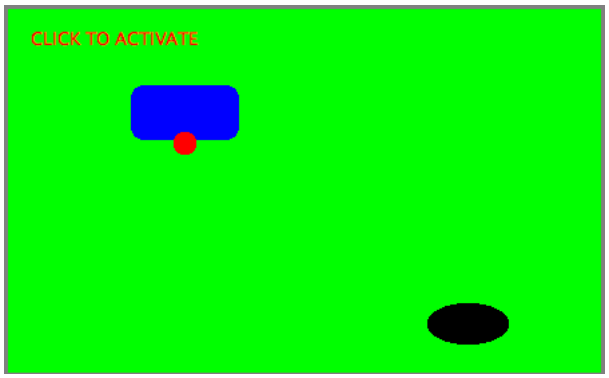
6.5.4 State Machines

The information stored in an object's instance variables is said to represent the ***state*** of that object. When one of the object's methods is called, the action taken by the object can depend on its state. (Or, in the terminology we have been using, the definition of the method can look at the instance variables to decide what to do.) Furthermore, the state can change. (That is, the definition of the method can assign new values to the instance variables.) In computer science, there is the idea of a ***state machine***, which is just something that has a state and can change state in response to events or inputs. The response of a state machine to an event or input depends on what state it's in. An object is a kind of state machine. Sometimes, this point of view can be very useful in designing classes.

The state machine point of view can be especially useful in the type of event-oriented programming that is required by graphical user interfaces. When designing a GUI program, you can ask yourself: What information about state do I need to keep track of? What events can change the state of the program? How will my response to a given event depend on the current state? Should the appearance of the GUI be changed to reflect a change in state? How should the `paintComponent()` method take the state into account? All this is an alternative to the top-down, step-wise-refinement style of program design, which does not apply to the overall design of an event-oriented program.

In the *KeyboardAndFocusDemo* program, shown above, the state of the program is recorded in the instance variables `squareColor`, `squareLeft`, and `squareTop`. These state variables are used in the `paintComponent()` method to decide how to draw the panel. Their values are changed in the two key-event-handling methods.

In the rest of this section, we'll look at another example, where the state plays an even bigger role. In this example, the user plays a simple arcade-style game by pressing the arrow keys. The main panel of the program is defined in the source code file *SubKillerPanel.java*. An applet that uses this panel can be found in *SubKillerApplet.java*, while the stand-alone application version is *SubKiller.java*. You can try out the applet in the on-line version of this section. Here is what it looks like:



You have to click on the panel to give it the input focus. The program shows a black "submarine" near the bottom of the panel. While the panel has the input focus, this submarine

moves back and forth erratically near the bottom. Near the top, there is a blue "boat." You can move this boat back and forth by pressing the left and right arrow keys. Attached to the boat is a red "bomb" (or "depth charge"). You can drop the bomb by hitting the down arrow key. The objective is to blow up the submarine by hitting it with the bomb. If the bomb falls off the bottom of the screen, you get a new one. If the submarine explodes, a new sub is created and you get a new bomb. Try it! Make sure to hit the sub at least once, so you can see the explosion.

Let's think about how this game can be programmed. First of all, since we are doing object-oriented programming, I decided to represent the boat, the depth charge, and the submarine as objects. Each of these objects is defined by a separate nested class inside the main panel class, and each object has its own state which is represented by the instance variables in the corresponding class. I use variables `boat`, `bomb`, and `sub` in the panel class to refer to the boat, bomb, and submarine objects.

Now, what constitutes the "state" of the program? That is, what things change from time to time and affect the appearance or behavior of the program? Of course, the state includes the positions of the boat, submarine, and bomb, so I need variables to store the positions. Anything else, possibly less obvious? Well, sometimes the bomb is falling, and sometimes it's not. That is a difference in state. Since there are two possibilities, I represent this aspect of the state with a boolean variable in the `bomb` object, `bomb.isFalling`. Sometimes the submarine is moving left and sometimes it is moving right. The difference is represented by another boolean variable, `sub.isMovingLeft`. Sometimes, the sub is exploding. This is also part of the state, and it is represented by a boolean variable, `sub.isExploding`. However, the explosions require a little more thought. An explosion is something that takes place over a series of frames. While an explosion is in progress, the sub looks different in each frame, as the size of the explosion increases. Also, I need to know when the explosion is over so that I can go back to moving and drawing the sub as usual. So, I use an integer variable, `sub.explosionFrameNumber` to record how many frames have been drawn since the explosion started; the value of this variable is used only when an explosion is in progress.

How and when do the values of these state variables change? Some of them seem to change on their own: For example, as the sub moves left and right, the state variables that specify its position change. Of course, these variables are changing because of an animation, and that animation is driven by a timer. Each time an event is generated by the timer, some of the state variables have to change to get ready for the next frame of the animation. The changes are made by the action listener that listens for events from the timer. The `boat`, `bomb`, and `sub` objects each contain an `updateForNextFrame()` method that updates the state variables of the object to get ready for the next frame of the animation. The action listener for the timer calls these methods with the statements

```
boat.updateForNewFrame();
bomb.updateForNewFrame();
sub.updateForNewFrame();
```

The action listener also calls `repaint()`, so that the panel will be redrawn to reflect its new state. There are several state variables that change in these update methods, in addition to the position of the sub: If the bomb is falling, then its y-coordinate increases from one frame to the next. If the bomb hits the sub, then the `isExploding` variable of the sub changes to true, and the `isFalling` variable of the bomb becomes `false`. The `isFalling` variable also becomes false when the bomb falls off the bottom of the screen. If the sub is exploding, then its `explosionFrameNumber` increases from one frame to the next, and when it reaches a certain

value, the explosion ends and `isExploding` is reset to false. At random times, the sub switches between moving to the left and moving to the right. Its direction of motion is recorded in the sub's `isMovingLeft` variable. The sub's `updateForNewFrame()` method includes the lines

```
if ( Math.random() < 0.04 )
   isMovingLeft = ! isMovingLeft;
```

There is a 1 in 25 chance that `Math.random()` will be less than 0.04, so the statement "`isMovingLeft = ! isMovingLeft`" is executed in one in every twenty-five frames, on average. The effect of this statement is to reverse the value of `isMovingLeft`, from false to true or from true to false. That is, the direction of motion of the sub is reversed.

In addition to changes in state that take place from one frame to the next, a few state variables change when the user presses certain keys. In the program, this is checked in a method that responds to user keystrokes. If the user presses the left or right arrow key, the position of the boat is changed. If the user presses the down arrow key, the bomb changes from not-falling to falling. This is coded in the `keyPressed()`method of a *KeyListener* that is registered to listen for key events on the panel; that method reads as follows:

```
public void keyPressed(KeyEvent evt) {
   int code = evt.getKeyCode();  // which key was pressed.
   if (code == KeyEvent.VK_LEFT) {
        // Move the boat left.  (If this moves the boat out of the frame, its
        // position will be adjusted in the boat.updateForNewFrame() method.)
      boat.centerX -= 15;
   }
   else if (code == KeyEvent.VK_RIGHT) {
        // Move the boat right.  (If this moves boat out of the frame, its
        // position will be adjusted in the boat.updateForNewFrame() method.)
      boat.centerX += 15;
   }
   else if (code == KeyEvent.VK_DOWN) {
         // Start the bomb falling, if it is not already falling.
      if ( bomb.isFalling == false )
         bomb.isFalling = true;
   }
}
```

Note that it's not necessary to call `repaint()` when the state changes, since this panel shows an animation that is constantly being redrawn anyway. Any changes in the state will become visible to the user as soon as the next frame is drawn. At some point in the program, I have to make sure that the user does not move the boat off the screen. I could have done this in `keyPressed()`, but I choose to check for this in another routine, in the boat object.

I encourage you to read the source code in *SubKillerPanel.java*. Although a few points are tricky, you should with some effort be able to read and understand the entire program. Try to understand the program in terms of state machines. Note how the state of each of the three objects in the program changes in response to events from the timer and from the user.

You should also note that the program uses four listeners, to respond to action events from the timer, key events from the user, focus events, and mouse events. (The mouse is used only to request the input focus when the user clicks the panel.) The timer runs only when the panel has the input focus; this is programmed by having the focus listener start the timer when the panel gains the input focus and stop the timer when the panel loses the input focus. All four listeners

are created in the constructor of the *SubKillerPanel* class using anonymous inner classes. (See Subsection 6.4.5.)

While it's not at all sophisticated as arcade games go, the SubKiller game does use some interesting programming. And it nicely illustrates how to apply state-machine thinking in event-oriented programming.

6.6 Basic Components

In preceding sections, you've seen how to use a graphics context to draw on the screen and how to handle mouse events and keyboard events. In one sense, that's all there is to GUI programming. If you're willing to program all the drawing and handle all the mouse and keyboard events, you have nothing more to learn. However, you would either be doing a lot more work than you need to do, or you would be limiting yourself to very simple user interfaces. A typical user interface uses standard GUI components such as buttons, scroll bars, text-input boxes, and menus. These components have already been written for you, so you don't have to duplicate the work involved in developing them. They know how to draw themselves, and they can handle the details of processing the mouse and keyboard events that concern them.

Consider one of the simplest user interface components, a push button. The button has a border, and it displays some text. This text can be changed. Sometimes the button is disabled, so that clicking on it doesn't have any effect. When it is disabled, its appearance changes. When the user clicks on the push button, the button changes appearance while the mouse button is pressed and changes back when the mouse button is released. In fact, it's more complicated than that. If the user moves the mouse outside the push button before releasing the mouse button, the button changes to its regular appearance. To implement this, it is necessary to respond to mouse exit or mouse drag events. Furthermore, on many platforms, a button can receive the input focus. The button changes appearance when it has the focus. If the button has the focus and the user presses the space bar, the button is triggered. This means that the button must respond to keyboard and focus events as well.

Fortunately, you don't have to program **any** of this, provided you use an object belonging to the standard class `javax.swing.JButton`. A *JButton* object draws itself and processes mouse, keyboard, and focus events on its own. You only hear from the *JButton* when the user triggers it by clicking on it or pressing the space bar while the button has the input focus. When this happens, the *JButton* object creates an event object belonging to the class `java.awt.event.ActionEvent`. The event object is sent to any registered listeners to tell them that the button has been pushed. Your program gets only the information it needs—the fact that a button was pushed.

* * *

The standard components that are defined as part of the Swing graphical user interface API are defined by subclasses of the class *JComponent*, which is itself a subclass of *Component*. (Note that this includes the *JPanel* class that we have already been working with extensively.) Many useful methods are defined in the *Component* and *JComponent* classes and so can be used with any Swing component. We begin by looking at a few of these methods. Suppose that `comp` is a variable that refers to some *JComponent*. Then the following methods can be used:

- `comp.getWidth()` and `comp.getHeight()` are functions that give the current size of the component, in pixels. One warning: When a component is first created, its size is zero. The size will be set later, probably by a layout manager. A common mistake is to check

the size of a component before that size has been set, such as in a constructor.

- `comp.setEnabled(true)` and `comp.setEnabled(false)` can be used to enable and disable the component. When a component is disabled, its appearance might change, and the user cannot do anything with it. There is a boolean-valued function, `comp.isEnabled()` that you can call to discover whether the component is enabled.
- `comp.setVisible(true)` and `comp.setVisible(false)` can be called to hide or show the component.
- `comp.setFont(font)` sets the font that is used for text displayed on the component. See Subsection 6.3.3 for a discussion of fonts.
- `comp.setBackground(color)` and `comp.setForeground(color)` set the background and foreground colors for the component. See Subsection 6.3.2.
- `comp.setOpaque(true)` tells the component that the area occupied by the component should be filled with the component's background color before the content of the component is painted. By default, only `JLabels` are non-opaque. A non-opaque, or "transparent", component ignores its background color and simply paints its content over the content of its container. This usually means that it inherits the background color from its container.
- `comp.setToolTipText(string)` sets the specified string as a "tool tip" for the component. The tool tip is displayed if the mouse cursor is in the component and the mouse is not moved for a few seconds. The tool tip should give some information about the meaning of the component or how to use it.
- `comp.setPreferredSize(size)` sets the size at which the component should be displayed, if possible. The parameter is of type `java.awt.Dimension`, where an object of type *Dimension* has two public integer-valued instance variables, `width` and `height`. A call to this method usually looks something like "`setPreferredSize( new Dimension(100,50) )`". The preferred size is used as a hint by layout managers, but will not be respected in all cases. Standard components generally compute a correct preferred size automatically, but it can be useful to set it in some cases. For example, if you use a `JPanel` as a drawing surface, it is usually a good idea to set a preferred size for it.

Note that using any component is a multi-step process. The component object must be created with a constructor. It must be added to a container. In many cases, a listener must be registered to respond to events from the component. And in some cases, a reference to the component must be saved in an instance variable so that the component can be manipulated by the program after it has been created. In this section, we will look at a few of the basic standard components that are available in Swing. In the next section we will consider the problem of laying out components in containers.

6.6.1 JButton

An object of class *JButton* is a push button that the user can click to trigger some action. You've already seen buttons used in Section 6.1 and Section 6.2, but we consider them in much more detail here. To use any component effectively, there are several aspects of the corresponding class that you should be familiar with. For *JButton*, as an example, I list these aspects explicitly:

- **Constructors**: The *JButton* class has a constructor that takes a string as a parameter. This string becomes the text displayed on the button. For example: `stopGoButton =`

`new JButton("Go")`. This creates a button object that will display the text, "Go" (but remember that the button must still be added to a container before it can appear on the screen).

- **Events**: When the user clicks on a button, the button generates an event of type *ActionEvent*. This event is sent to any listener that has been registered with the button as an *ActionListener*.
- **Listeners**: An object that wants to handle events generated by buttons must implement the *ActionListener* interface. This interface defines just one method, "`public void actionPerformed(ActionEvent evt)`", which is called to notify the object of an action event.
- **Registration of Listeners**: In order to actually receive notification of an event from a button, an *ActionListener* must be registered with the button. This is done with the button's `addActionListener()` method. For example: `stopGoButton.addActionListener( buttonHandler );`
- **Event methods**: When `actionPerformed(evt)` is called by the button, the parameter, `evt`, contains information about the event. This information can be retrieved by calling methods in the *ActionEvent* class. In particular, `evt.getActionCommand()` returns a *String* giving the command associated with the button. By default, this command is the text that is displayed on the button, but it is possible to set it to some other string. The method `evt.getSource()` returns a reference to the `Object` that produced the event, that is, to the *JButton* that was pressed. The return value is of type `Object`, not *JButton*, because other types of components can also produce *ActionEvents*.
- **Component methods**: Several useful methods are defined in the *JButton* class. For example, `stopGoButton.setText("Stop")` changes the text displayed on the button to "Stop". And `stopGoButton.setActionCommand("sgb")` changes the action command associated with this button for action events.

Of course, `JButtons` also have all the general `Component` methods, such as `setEnabled()` and `setFont()`. The `setEnabled()` and `setText()` methods of a button are particularly useful for giving the user information about what is going on in the program. A disabled button is better than a button that gives an obnoxious error message such as "Sorry, you can't click on me now!"

6.6.2 JLabel

JLabel is certainly the simplest type of component. An object of type *JLabel* exists just to display a line of text. The text cannot be edited by the user, although it can be changed by your program. The constructor for a *JLabel* specifies the text to be displayed:

```
JLabel message = new JLabel("Hello World!");
```

There is another constructor that specifies where in the label the text is located, if there is extra space. The possible alignments are given by the constants `JLabel.LEFT`, `JLabel.CENTER`, and `JLabel.RIGHT`. For example,

```
JLabel message = new JLabel("Hello World!", JLabel.CENTER);
```

creates a label whose text is centered in the available space. You can change the text displayed in a label by calling the label's `setText()` method:

```
message.setText("Goodbye World!");
```

Since the *JLabel* class is a subclass of *JComponent*, you can use methods such as `setForeground()` and `setFont()` with labels. If you want the background color to have any effect, you should call `setOpaque(true)` on the label, since otherwise the *JLabel* might not fill in its background. For example:

```
JLabel message = new JLabel("Hello World!", JLabel.CENTER);
message.setForeground(Color.RED);   // Display red text...
message.setBackground(Color.BLACK); //    on a black background...
message.setFont(new Font("Serif",Font.BOLD,18));  // in a big bold font.
message.setOpaque(true);  // Make sure background is filled in.
```

6.6.3 JCheckBox

A *JCheckBox* is a component that has two states: selected or unselected. The user can change the state of a check box by clicking on it. The state of a checkbox is represented by a **boolean** value that is `true` if the box is selected and is `false` if the box is unselected. A checkbox has a label, which is specified when the box is constructed:

```
JCheckBox showTime = new JCheckBox("Show Current Time");
```

Usually, it's the user who sets the state of a *JCheckBox*, but you can also set the state in your program. The current state of a checkbox is set using its `setSelected(boolean)` method. For example, if you want the checkbox `showTime` to be checked, you would say "`showTime.setSelected(true)`". To uncheck the box, say "`showTime.setSelected(false)`". You can determine the current state of a checkbox by calling its `isSelected()` method, which returns a boolean value.

In many cases, you don't need to worry about events from checkboxes. Your program can just check the state whenever it needs to know it by calling the `isSelected()` method. However, a checkbox does generate an event when its state is changed by the user, and you can detect this event and respond to it if you want something to happen at the moment the state changes. When the state of a checkbox is changed by the user, it generates an event of type *ActionEvent*. If you want something to happen when the user changes the state, you must register an *ActionListener* with the checkbox by calling its `addActionListener()` method. (Note that if you change the state by calling the `setSelected()` method, no *ActionEvent* is generated. However, there is another method in the *JCheckBox* class, `doClick()`, which simulates a user click on the checkbox and does generate an *ActionEvent*.)

When handling an *ActionEvent*, you can call `evt.getSource()` in the `actionPerformed()` method to find out which object generated the event. (Of course, if you are only listening for events from one component, you don't have to do this.) The returned value is of type `Object`, but you can type-cast it to another type if you want. Once you know the object that generated the event, you can ask the object to tell you its current state. For example, if you know that the event had to come from one of two checkboxes, `cb1` or `cb2`, then your `actionPerformed()` method might look like this:

```
public void actionPerformed(ActionEvent evt) {
   Object source = evt.getSource();
   if (source == cb1) {
      boolean newState = cb1.isSelected();
      ... // respond to the change of state
   }
   else if (source == cb2) {
```

```
            boolean newState = cb2.isSelected();
            ... // respond to the change of state
        }
    }
```

Alternatively, you can use `evt.getActionCommand()` to retrieve the action command associated with the source. For a *JCheckBox*, the action command is, by default, the label of the checkbox.

6.6.4 JTextField and JTextArea

The *JTextField* and *JTextArea* classes represent components that contain text that can be edited by the user. A *JTextField* holds a single line of text, while a *JTextArea* can hold multiple lines. It is also possible to set a *JTextField* or *JTextArea* to be read-only so that the user can read the text that it contains but cannot edit the text. Both classes are subclasses of an abstract class, *JTextComponent*, which defines their common properties.

JTextField and *JTextArea* have many methods in common. The instance method `setText()`, which takes a parameter of type *String*, can be used to change the text that is displayed in an input component. The contents of the component can be retrieved by calling its `getText()` instance method, which returns a value of type *String*. If you want to stop the user from modifying the text, you can call `setEditable(false)`. Call the same method with a parameter of `true` to make the input component user-editable again.

The user can only type into a text component when it has the input focus. The user can give the input focus to a text component by clicking it with the mouse, but sometimes it is useful to give the input focus to a text field programmatically. You can do this by calling its `requestFocus()` method. For example, when I discover an error in the user's input, I usually call `requestFocus()` on the text field that contains the error. This helps the user see where the error occurred and lets the user start typing the correction immediately.

By default, there is no space between the text in a text component and the edge of the component, which usually doesn't look very good. You can use the `setMargin()` method of the component to add some blank space between the edge of the component and the text. This method takes a parameter of type `java.awt.Insets` which contains four integer instance variables that specify the margins on the top, left, bottom, and right edge of the component. For example,

```
textComponent.setMargin( new Insets(5,5,5,5) );
```

adds a five-pixel margin between the text in `textComponent` and each edge of the component.

* * *

The *JTextField* class has a constructor

```
public JTextField(int columns)
```

where `columns` is an integer that specifies the number of characters that should be visible in the text field. This is used to determine the preferred width of the text field. (Because characters can be of different sizes and because the preferred width is not always respected, the actual number of characters visible in the text field might not be equal to `columns`.) You don't have to specify the number of columns; for example, you might use the text field in a context where it will expand to fill whatever space is available. In that case, you can use the default constructor `JTextField()`, with no parameters. You can also use the following constructors, which specify the initial contents of the text field:

```
public JTextField(String contents);
public JTextField(String contents, int columns);
```

The constructors for a *JTextArea* are

```
public JTextArea()
public JTextArea(int rows, int columns)
public JTextArea(String contents)
public JTextArea(String contents, int rows, int columns)
```

The parameter `rows` specifies how many lines of text should be visible in the text area. This determines the preferred height of the text area, just as `columns` determines the preferred width. However, the text area can actually contain any number of lines; the text area can be scrolled to reveal lines that are not currently visible. It is common to use a *JTextArea* as the `CENTER` component of a BorderLayout. In that case, it is less useful to specify the number of lines and columns, since the TextArea will expand to fill all the space available in the center area of the container.

The *JTextArea* class adds a few useful methods to those inherited from `JTextComponent`. For example, the instance method `append(moreText)`, where `moreText` is of type *String*, adds the specified text at the end of the current content of the text area. (When using `append()` or `setText()` to add text to a *JTextArea*, line breaks can be inserted in the text by using the newline character, `'\n'`.) And `setLineWrap(wrap)`, where `wrap` is of type **boolean**, tells what should happen when a line of text is too long to be displayed in the text area. If `wrap` is true, then any line that is too long will be "wrapped" onto the next line; if `wrap` is false, the line will simply extend outside the text area, and the user will have to scroll the text area horizontally to see the entire line. The default value of `wrap` is false.

Since it might be necessary to scroll a text area to see all the text that it contains, you might expect a text area to come with scroll bars. Unfortunately, this does not happen automatically. To get scroll bars for a text area, you have to put the *JTextArea* inside another component, called a *JScrollPane*. This can be done as follows:

```
JTextArea inputArea = new JTextArea();
JScrollPane scroller = new JScrollPane( inputArea );
```

The scroll pane provides scroll bars that can be used to scroll the text in the text area. The scroll bars will appear only when needed, that is when the size of the text exceeds the size of the text area. Note that when you want to put the text area into a container, you should add the scroll pane, not the text area itself, to the container. See the short example *TextAreaDemo.java* for an example of using a text area.

* * *

When the user is typing in a *JTextField* and presses return, an *ActionEvent* is generated. If you want to respond to such events, you can register an *ActionListener* with the text field, using the text field's `addActionListener()` method. (Since a *JTextArea* can contain multiple lines of text, pressing return in a text area does not generate an event; it simply begins a new line of text.)

JTextField has a subclass, *JPasswordField*, which is identical except that it does not reveal the text that it contains. The characters in a *JPasswordField* are all displayed as asterisks (or some other fixed character). A password field is, obviously, designed to let the user enter a password without showing that password on the screen.

Text components are actually quite complex, and I have covered only their most basic properties here. I will return to the topic of text components in Subsection 13.4.4.

6.6.5 JComboBox

The *JComboBox* class provides a way to let the user select one option from a list of options. The options are presented as a kind of pop-up menu, and only the currently selected option is visible on the screen.

When a *JComboBox* object is first constructed, it initially contains no items. An item is added to the bottom of the list of options by calling the combo box's instance method, `addItem(str)`, where `str` is the string that will be displayed in the menu.

For example, the following code will create an object of type *JComboBox* that contains the options Red, Blue, Green, and Black:

```
JComboBox colorChoice = new JComboBox();
colorChoice.addItem("Red");
colorChoice.addItem("Blue");
colorChoice.addItem("Green");
colorChoice.addItem("Black");
```

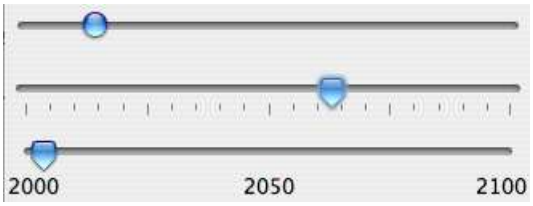
You can call the `getSelectedIndex()` method of a *JComboBox* to find out which item is currently selected. This method returns an integer that gives the position of the selected item in the list, where the items are numbered starting from zero. Alternatively, you can call `getSelectedItem()` to get the selected item itself. (This method returns a value of type *Object*, since a *JComboBox* can actually hold other types of objects besides strings.) You can change the selection by calling the method `setSelectedIndex(n)`, where `n` is an integer giving the position of the item that you want to select.

The most common way to use a *JComboBox* is to call its `getSelectedIndex()` method when you have a need to know which item is currently selected. However, like other components that we have seen, *JComboBox* components generate *ActionEvents* when the user selects an item. You can register an *ActionListener* with the *JComboBox* if you want to respond to such events as they occur.

JComboBoxes have a nifty feature, which is probably not all that useful in practice. You can make a *JComboBox* "editable" by calling its method `setEditable(true)`. If you do this, the user can edit the selection by clicking on the *JComboBox* and typing. This allows the user to make a selection that is not in the pre-configured list that you provide. (The "Combo" in the name "JComboBox" refers to the fact that it's a kind of combination of menu and text-input box.) If the user has edited the selection in this way, then the `getSelectedIndex()` method will return the value `-1`, and `getSelectedItem()` will return the string that the user typed. An *ActionEvent* is triggered if the user presses return while typing in the *JComboBox*.

6.6.6 JSlider

A *JSlider* provides a way for the user to select an integer value from a range of possible values. The user does this by dragging a "knob" along a bar. A slider can, optionally, be decorated with tick marks and with labels. This picture shows three sliders with different decorations and with different ranges of values:



Here, the second slider is decorated with ticks, and the third one is decorated with labels. It's possible for a single slider to have both types of decorations.

The most commonly used constructor for `JSliders` specifies the start and end of the range of values for the slider and its initial value when it first appears on the screen:

```
public JSlider(int minimum, int maximum, int value)
```

If the parameters are omitted, the values 0, 100, and 50 are used. By default, a slider is horizontal, but you can make it vertical by calling its method `setOrientation(JSlider.VERTICAL)`. The current value of a *JSlider* can be read at any time with its `getValue()` method, which returns a value of type **int**. If you want to change the value, you can do so with the method `setValue(n)`, which takes a parameter of type **int**.

If you want to respond immediately when the user changes the value of a slider, you can register a listener with the slider. `JSliders`, unlike other components we have seen, do not generate `ActionEvents`. Instead, they generate events of type *ChangeEvent*. *ChangeEvent* and related classes are defined in the package `javax.swing.event` rather than `java.awt.event`, so if you want to use `ChangeEvents`, you should `import javax.swing.event.*` at the beginning of your program. You must also define some object to implement the *ChangeListener* interface, and you must register the change listener with the slider by calling its `addChangeListener()` method. A *ChangeListener* must provide a definition for the method:

```
public void stateChanged(ChangeEvent evt)
```

This method will be called whenever the value of the slider changes. Note that it will also be called when you change the value with the `setValue()` method, as well as when the user changes the value. In the `stateChanged()` method, you can call `evt.getSource()` to find out which object generated the event. If you want to know whether the user generated the change event, call the slider's `getValueIsAdjusting()` method, which returns true if the user is dragging the knob on the slider.

Using tick marks on a slider is a two-step process: Specify the interval between the tick marks, and tell the slider that the tick marks should be displayed. There are actually two types of tick marks, "major" tick marks and "minor" tick marks. You can have one or the other or both. Major tick marks are a bit longer than minor tick marks. The method `setMinorTickSpacing(i)` indicates that there should be a minor tick mark every `i` units along the slider. The parameter is an integer. (The spacing is in terms of values on the slider, not pixels.) For the major tick marks, there is a similar command, `setMajorTickSpacing(i)`. Calling these methods is not enough to make the tick marks appear. You also have to call `setPaintTicks(true)`. For example, the second slider in the above picture was created and configured using the commands:

```
slider2 = new JSlider();  // (Uses default min, max, and value.)
slider2.addChangeListener(this);
slider2.setMajorTickSpacing(25);
slider2.setMinorTickSpacing(5);
slider2.setPaintTicks(true);
```

Labels on a slider are handled similarly. You have to specify the labels and tell the slider to paint them. Specifying labels is a tricky business, but the *JSlider* class has a method to simplify it. You can create a set of labels and add them to a slider named `sldr` with the command:

```
sldr.setLabelTable( sldr.createStandardLabels(i) );
```

where `i` is an integer giving the spacing between the labels. To arrange for the labels to be displayed, call `setPaintLabels(true)`. For example, the third slider in the above picture was created and configured with the commands:

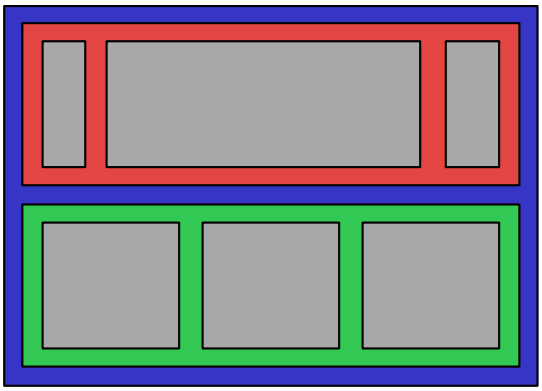
```
slider3 = new JSlider(2000,2100,2006);
slider3.addChangeListener(this);
slider3.setLabelTable( slider3.createStandardLabels(50) );
slider3.setPaintLabels(true);
```

6.7 Basic Layout

Components are the fundamental building blocks of a graphical user interface. But you have to do more with components besides create them. Another aspect of GUI programming is ***laying out*** components on the screen, that is, deciding where they are drawn and how big they are. You have probably noticed that computing coordinates can be a difficult problem, especially if you don't assume a fixed size for the drawing area. Java has a solution for this, as well.

Components are the visible objects that make up a GUI. Some components are ***containers***, which can hold other components. Containers in Java are objects that belong to some subclass of `java.awt.Container`. The content pane of a *JApplet* or *JFrame* is an example of a container. The standard class *JPanel*, which we have mostly used as a drawing surface up until now, is another example of a container.

Because a *JPanel* object is a container, it can hold other components. Because a *JPanel* is itself a component, you can add a *JPanel* to another *JPanel*. This makes complex nesting of components possible. *JPanels* can be used to organize complicated user interfaces, as shown in this illustration:



Three panels, shown in color, containing six other components, shown in gray.

The components in a container must be "laid out," which means setting their sizes and positions. It's possible to program the layout yourself, but layout is ordinarily done by a ***layout manager***. A layout manager is an object associated with a container that implements some policy for laying out the components in that container. Different types of layout manager

implement different policies. In this section, we will cover the three most common types of layout manager, and then we will look at several programming examples that use components and layout.

Every container has a default layout manager and has an instance method, `setLayout()`, that takes a parameter of type *LayoutManager* and that is used to specify a different layout manager for the container. Components are added to a container by calling an instance method named `add()` in the container object. There are actually several versions of the `add()` method, with different parameter lists. Different versions of `add()` are appropriate for different layout managers, as we will see below.

6.7.1 Basic Layout Managers

Java has a variety of standard layout managers that can be used as parameters in the `setLayout()` method. They are defined by classes in the package `java.awt`. Here, we will look at just three of these layout manager classes: *FlowLayout*, *BorderLayout*, and *GridLayout*.

A *FlowLayout* simply lines up components in a row across the container. The size of each component is equal to that component's "preferred size." After laying out as many items as will fit in a row across the container, the layout manager will move on to the next row. The default layout for a *JPanel* is a *FlowLayout*; that is, a *JPanel* uses a *FlowLayout* unless you specify a different layout manager by calling the panel's `setLayout()` method.

The components in a given row can be either left-aligned, right-aligned, or centered within that row, and there can be horizontal and vertical gaps between components. If the default constructor, "`new FlowLayout()`", is used, then the components on each row will be centered and both the horizontal and the vertical gaps will be five pixels. The constructor

```
public FlowLayout(int align, int hgap, int vgap)
```

can be used to specify alternative alignment and gaps. The possible values of `align` are `FlowLayout.LEFT`, `FlowLayout.RIGHT`, and `FlowLayout.CENTER`.

Suppose that `cntr` is a container object that is using a *FlowLayout* as its layout manager. Then, a component, `comp`, can be added to the container with the statement

```
cntr.add(comp);
```

The *FlowLayout* will line up all the components that have been added to the container in this way. They will be lined up in the order in which they were added. For example, this picture shows five buttons in a panel that uses a *FlowLayout*:

Note that since the five buttons will not fit in a single row across the panel, they are arranged in two rows. In each row, the buttons are grouped together and are centered in the row. The buttons were added to the panel using the statements:

```
panel.add(button1);
panel.add(button2);
panel.add(button3);
panel.add(button4);
panel.add(button5);
```

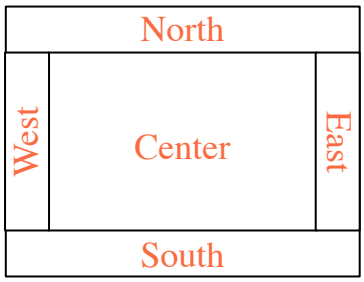
When a container uses a layout manager, the layout manager is ordinarily responsible for computing the preferred size of the container (although a different preferred size could be set by calling the container's `setPreferredSize` method). A *FlowLayout* prefers to put its components in a single row, so the preferred width is the total of the preferred widths of all the components, plus the horizontal gaps between the components. The preferred height is the maximum preferred height of all the components.

* * *

A *BorderLayout* layout manager is designed to display one large, central component, with up to four smaller components arranged along the edges of the central component. If a container, `cntr`, is using a *BorderLayout*, then a component, `comp`, should be added to the container using a statement of the form

```
cntr.add( comp, borderLayoutPosition );
```

where `borderLayoutPosition` specifies what position the component should occupy in the layout and is given as one of the constants `BorderLayout.CENTER`, `BorderLayout.NORTH`, `BorderLayout.SOUTH`, `BorderLayout.EAST`, or `BorderLayout.WEST`. The meaning of the five positions is shown in this diagram:



Note that a border layout can contain fewer than five components, so that not all five of the possible positions need to be filled. It would be very unusual, however, to have no center component.

A *BorderLayout* selects the sizes of its components as follows: The `NORTH` and `SOUTH` components (if present) are shown at their preferred heights, but their width is set equal to the full width of the container. The `EAST` and `WEST` components are shown at their preferred widths, but their height is set to the height of the container, minus the space occupied by the `NORTH` and `SOUTH` components. Finally, the `CENTER` component takes up any remaining space. The preferred size of the `CENTER` component is ignored when the layout is done, but it is taken into account when the preferred size of the container is computed. You should make sure that the components that you put into a *BorderLayout* are suitable for the positions that they will occupy. A horizontal slider or text field, for example, would work well in the `NORTH` or `SOUTH` position, but wouldn't make much sense in the `EAST` or `WEST` position.

The default constructor, `new BorderLayout()`, leaves no space between components. If you would like to leave some space, you can specify horizontal and vertical gaps in the constructor of the `BorderLayout` object. For example, if you say

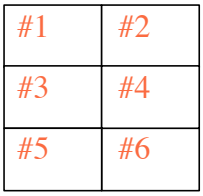
```
panel.setLayout(new BorderLayout(5,7));
```

then the layout manager will insert horizontal gaps of 5 pixels between components and vertical gaps of 7 pixels between components. The background color of the container will show through

in these gaps. The default layout for the original content pane that comes with a *JFrame* or *JApplet* is a `BorderLayout` with no horizontal or vertical gap.

* * *

Finally, we consider the *GridLayout* layout manager. A grid layout lays out components in a grid containing rows and columns of equal sized rectangles. This illustration shows how the components would be arranged in a grid layout with 3 rows and 2 columns:



If a container uses a *GridLayout*, the appropriate `add` method for the container takes a single parameter of type *Component* (for example: `cntr.add(comp)`). Components are added to the grid in the order shown; that is, each row is filled from left to right before going on the next row.

The constructor for a *GridLayout* takes the form "`new GridLayout(R,C)`", where `R` is the number of rows and `C` is the number of columns. If you want to leave horizontal gaps of `H` pixels between columns and vertical gaps of `V` pixels between rows, use "`new GridLayout(R,C,H,V)`" instead.

When you use a *GridLayout*, it's probably good form to add just enough components to fill the grid. However, this is not required. In fact, as long as you specify a non-zero value for the number of rows, then the number of columns is essentially ignored. The system will use just as many columns as are necessary to hold all the components that you add to the container. If you want to depend on this behavior, you should probably specify zero as the number of columns. You can also specify the number of rows as zero. In that case, you must give a non-zero number of columns. The system will use the specified number of columns, with just as many rows as necessary to hold the components that are added to the container.

Horizontal grids, with a single row, and vertical grids, with a single column, are very common. For example, suppose that `button1`, `button2`, and `button3` are buttons and that you'd like to display them in a horizontal row in a panel. If you use a horizontal grid for the panel, then the buttons will completely fill that panel and will all be the same size. The panel can be created as follows:

```
JPanel buttonBar = new JPanel();
buttonBar.setLayout( new GridLayout(1,3) );
    // (Note:  The "3" here is pretty much ignored, and
    //  you could also say "new GridLayout(1,0)".
    //  To leave gaps between the buttons, you could use
    //  "new GridLayout(1,0,5,5)".)
buttonBar.add(button1);
buttonBar.add(button2);
buttonBar.add(button3);
```

You might find this button bar to be more attractive than the one that uses the default *FlowLayout* layout manager.

6.7.2 Borders

We have seen how to leave gaps between the components in a container, but what if you would like to leave a border around the outside of the container? This problem is not handled by layout managers. Instead, borders in Swing are represented by objects. A *Border* object can be added to any *JComponent*, not just to containers. Borders can be more than just empty space. The class `javax.swing.BorderFactory` contains a large number of static methods for creating border objects. For example, the function

```
BorderFactory.createLineBorder(Color.BLACK)
```

returns an object that represents a one-pixel wide black line around the outside of a component. If `comp` is a `JComponent`, a border can be added to `comp` using its `setBorder()` method. For example:

```
comp.setBorder( BorderFactory.createLineBorder(Color.BLACK) );
```

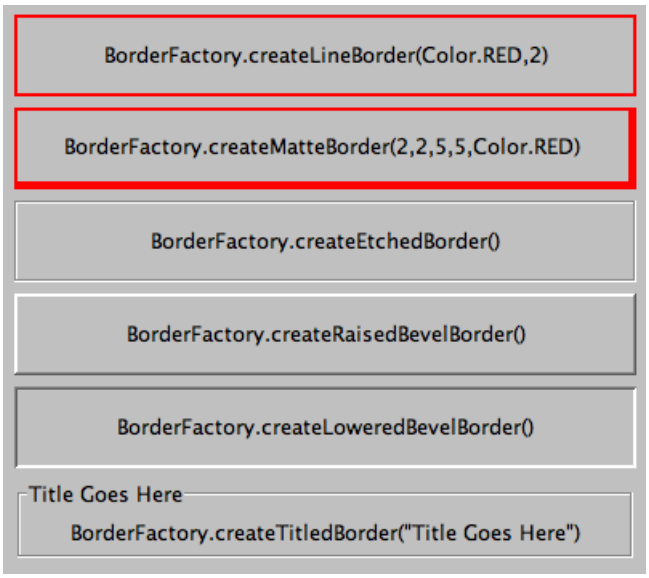
Once a border has been set for a `JComponent`, the border is drawn automatically, without any further effort on the part of the programmer. The border is drawn along the edges of the component, just inside its boundary. The layout manager of a `JPanel` or other container will take the space occupied by the border into account. The components that are added to the container will be displayed in the area inside the border. I don't recommend using a border on a `JPanel` that is being used as a drawing surface. However, if you do this, you should take the border into account. If you draw in the area occupied by the border, that part of your drawing will be covered by the border.

Here are some of the static methods that can be used to create borders:

- `BorderFactory.createEmptyBorder(top,left,bottom,right)` — leaves an empty border around the edges of a component. Nothing is drawn in this space, so the background color of the component will appear in the area occupied by the border. The parameters are integers that give the width of the border along the top, left, bottom, and right edges of the component. This is actually very useful when used on a *JPanel* that contains other components. It puts some space between the components and the edge of the panel. It can also be useful on a *JLabel*, which otherwise would not have any space between the text and the edge of the label.
- `BorderFactory.createLineBorder(color,thickness)` — draws a line around all four edges of a component. The first parameter is of type *Color* and specifies the color of the line. The second parameter is an integer that specifies the thickness of the border, in pixels. If the second parameter is omitted, a line of thickness 1 is drawn.
- `BorderFactory.createMatteBorder(top,left,bottom,right,color)` — is similar to `createLineBorder`, except that you can specify individual thicknesses for the top, left, bottom, and right edges of the component.
- `BorderFactory.createEtchedBorder()` — creates a border that looks like a groove etched around the boundary of the component. The effect is achieved using lighter and darker shades of the component's background color, and it does not work well with every background color.
- `BorderFactory.createLoweredBevelBorder()`—gives a component a three-dimensional effect that makes it look like it is lowered into the computer screen. As with an EtchedBorder, this only works well for certain background colors.

- `BorderFactory.createRaisedBevelBorder()`—similar to a LoweredBevelBorder, but the component looks like it is raised above the computer screen.
- `BorderFactory.createTitledBorder(title)`—creates a border with a title. The title is a *String*, which is displayed in the upper left corner of the border.

There are many other methods in the `BorderFactory` class, most of them providing variations of the basic border styles given here. The following illustration shows six components with six different border styles. The text in each component is the command that created the border for that component:



(The source code for the applet that produced this picture can be found in *BorderDemo.java*.)

6.7.3 SliderAndComboBoxDemo

Now that we have looked at components and layouts, it's time to put them together into some complete programs. We start with a simple demo that uses a *JLabel*, a *JComboBox*, and a couple of *JSliders*, all laid out in a *GridLayout*, as shown in this picture:

The sliders in this applet control the foreground and background color of the label, and the combo box controls its font style. Writing this program is a matter of creating the components, laying them out, and programming listeners to respond to events from the sliders and combo box. In my program, I define a subclass of *JPanel* which will be used for the applet's content pane. This class implements *ChangeListener* and *ActionListener*, so the panel itself can act as the listener for change events from the sliders and action events from the combo box. In the

constructor, the four components are created and configured, a *GridLayout* is installed as the layout manager for the panel, and the components are added to the panel:

```
/* Create the sliders, and set up this panel to listen for
   ChangeEvents that are generated by the sliders. */

bgColorSlider = new JSlider(0,255,100);
bgColorSlider.addChangeListener(this);

fgColorSlider = new JSlider(0,255,200);
fgColorSlider.addChangeListener(this);

/* Create the combo box, and add four items to it, listing
   different font styles.  Set up the panel to listen for
   ActionEvents from the combo box. */

fontStyleSelect = new JComboBox();
fontStyleSelect.addItem("Plain Font");
fontStyleSelect.addItem("Italic Font");
fontStyleSelect.addItem("Bold Font");
fontStyleSelect.addItem("Bold Italic Font");
fontStyleSelect.setSelectedIndex(2);
fontStyleSelect.addActionListener(this);

/* Create the display label, with properties to match the
   values of the sliders and the setting of the combo box. */

displayLabel = new JLabel("Hello World!", JLabel.CENTER);
displayLabel.setOpaque(true);
displayLabel.setBackground( new Color(100,100,100) );
displayLabel.setForeground( new Color(255, 200, 200) );
displayLabel.setFont( new Font("Serif", Font.BOLD, 30) );

/* Set the layout for the panel, and add the four components.
   Use a GridLayout with 4 rows and 1 column. */

setLayout(new GridLayout(4,1));
add(displayLabel);
add(bgColorSlider);
add(fgColorSlider);
add(fontStyleSelect);
```

The class also defines the methods required by the *ActionListener* and *ChangeListener* interfaces. The `actionPerformed()` method is called when the user selects an item in the combo box. This method changes the font in the *JLabel*, where the font depends on which item is currently selected in the combo box, `fontStyleSelect`:

```
public void actionPerformed(ActionEvent evt) {
   switch ( fontStyleSelect.getSelectedIndex() ) {
   case 0:
      displayLabel.setFont( new Font("Serif", Font.PLAIN, 30) );
      break;
   case 1:
      displayLabel.setFont( new Font("Serif", Font.ITALIC, 30) );
      break;
   case 2:
      displayLabel.setFont( new Font("Serif", Font.BOLD, 30) );
      break;
```

```
            case 3:
               displayLabel.setFont( new Font("Serif", Font.BOLD + Font.ITALIC, 30) );
               break;
            }
         }
```

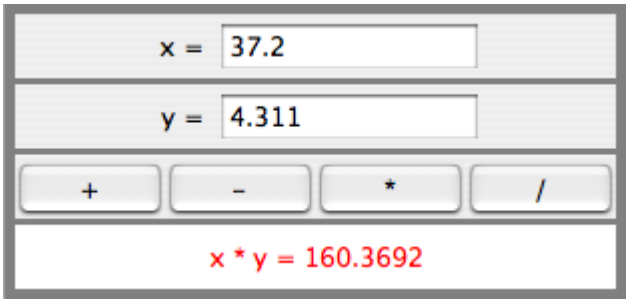
And the `stateChanged()` method, which is called when the user manipulates one of the sliders, uses the value on the slider to compute a new foreground or background color for the label. The method checks `evt.getSource()` to determine which slider was changed:

```
      public void stateChanged(ChangeEvent evt) {
         if (evt.getSource() == bgColorSlider) {
            int bgVal = bgColorSlider.getValue();
            displayLabel.setBackground( new Color(bgVal,bgVal,bgVal) );
               // NOTE:  The background color is a shade of gray,
               //        determined by the setting on the slider.
         }
         else {
            int fgVal = fgColorSlider.getValue();
            displayLabel.setForeground( new Color( 255, fgVal, fgVal) );
               // Note:  The foreground color ranges from pure red to pure
               //        white as the slider value increases from 0 to 255.
         }
      }
```

(The complete source code is in the file *SliderAndComboBoxDemo.java*.)

6.7.4 A Simple Calculator

As our next example, we look briefly at an example that uses nested subpanels to build a more complex user interface. The program has two *JTextFields* where the user can enter two numbers, four *JButtons* that the user can click to add, subtract, multiply, or divide the two numbers, and a *JLabel* that displays the result of the operation:



Like the previous example, this example uses a main panel with a *GridLayout* that has four rows and one column. In this case, the layout is created with the statement:

```
setLayout(new GridLayout(4,1,3,3));
```

which allows a 3-pixel gap between the rows where the gray background color of the panel is visible. The gray border around the edges of the panel is added with the statement

```
setBorder( BorderFactory.createEmptyBorder(5,5,5,5) );
```

The first row of the grid layout actually contains two components, a *JLabel* displaying the text "x =" and a *JTextField*. A grid layout can only have one component in each position. In this case, that component is a *JPanel*, a subpanel that is nested inside the main panel. This subpanel in turn contains the label and text field. This can be programmed as follows:

```
xInput = new JTextField("0", 10); // Create a text field sized to hold 10 chars.
JPanel xPanel = new JPanel();     // Create the subpanel.
xPanel.add( new JLabel(" x = ")); // Add a label to the subpanel.
xPanel.add(xInput);               // Add the text field to the subpanel
mainPanel.add(xPanel);            // Add the subpanel to the main panel.
```

The subpanel uses the default *FlowLayout* layout manager, so the label and text field are simply placed next to each other in the subpanel at their preferred size, and are centered in the subpanel.

Similarly, the third row of the grid layout is a subpanel that contains four buttons. In this case, the subpanel uses a *GridLayout* with one row and four columns, so that the buttons are all the same size and completely fill the subpanel.

One other point of interest in this example is the `actionPerformed()` method that responds when the user clicks one of the buttons. This method must retrieve the user's numbers from the text field, perform the appropriate arithmetic operation on them (depending on which button was clicked), and set the text of the label (named `answer`) to represent the result. However, the contents of the text fields can only be retrieved as strings, and these strings must be converted into numbers. If the conversion fails, the label is set to display an error message:

```
public void actionPerformed(ActionEvent evt) {

   double x, y;  // The numbers from the input boxes.

   try {
      String xStr = xInput.getText();
      x = Double.parseDouble(xStr);
   }
   catch (NumberFormatException e) {
          // The string xStr is not a legal number.
      answer.setText("Illegal data for x.");
      xInput.requestFocus();
      return;
   }

   try {
      String yStr = yInput.getText();
      y = Double.parseDouble(yStr);
   }
   catch (NumberFormatException e) {
         // The string yStr is not a legal number.
      answer.setText("Illegal data for y.");
      yInput.requestFocus();
      return;
   }

   /* Perform the operation based on the action command from the
    button.  The action command is the text displayed on the button.
    Note that division by zero produces an error message. */

   String op = evt.getActionCommand();
```

```
        if (op.equals("+"))
           answer.setText( "x + y = " + (x+y) );
        else if (op.equals("-"))
           answer.setText( "x - y = " + (x-y) );
        else if (op.equals("*"))
           answer.setText( "x * y = " + (x*y) );
        else if (op.equals("/")) {
           if (y == 0)
              answer.setText("Can't divide by zero!");
           else
              answer.setText( "x / y = " + (x/y) );
        }

     } // end actionPerformed()
```

(The complete source code for this example can be found in *SimpleCalc.java*.)

6.7.5 Using a null Layout

As mentioned above, it is possible to do without a layout manager altogether. For our next example, we'll look at a panel that does not use a layout manager. If you set the layout manager of a container to be `null`, by calling `container.setLayout(null)`, then you assume complete responsibility for positioning and sizing the components in that container.

If `comp` is any component, then the statement

```
comp.setBounds(x, y, width, height);
```

puts the top left corner of the component at the point `(x,y)`, measured in the coordinate system of the container that contains the component, and it sets the width and height of the component to the specified values. You should only set the bounds of a component if the container that contains it has a null layout manager. In a container that has a non-null layout manager, the layout manager is responsible for setting the bounds, and you should not interfere with its job.

Assuming that you have set the layout manager to `null`, you can call the `setBounds()` method any time you like. (You can even make a component that moves or changes size while the user is watching.) If you are writing a panel that has a known, fixed size, then you can set the bounds of each component in the panel's constructor. Note that you must also add the components to the panel, using the panel's `add(component)` instance method; otherwise, the component will not appear on the screen.

Our example contains four components: two buttons, a label, and a panel that displays a checkerboard pattern:

This is just an example of using a null layout; it doesn't do anything, except that clicking the buttons changes the text of the label. (We will use this example in Section 7.5 as a starting point for a checkers game.)

For its content pane, this example uses a main panel that is defined by a class named *NullLayoutPanel*. The four components are created and added to the panel in the constructor of the *NullLayoutPanel* class. Then the `setBounds()` method of each component is called to set the size and position of the component:

```
public NullLayoutPanel() {

   setLayout(null);  // I will do the layout myself!

   setBackground(new Color(0,150,0));  // A dark green background.

   setBorder( BorderFactory.createEtchedBorder() );

   setPreferredSize( new Dimension(350,240) );
        // I assume that the size of the panel is, in fact, 350-by-240.

   /* Create the components and add them to the content pane.  If you
    don't add them to a container, they won't appear, even if
    you set their bounds! */

   board = new Checkerboard();
       // (Checkerboard is a subclass of JPanel, defined elsewhere.)
   add(board);

   newGameButton = new JButton("New Game");
   newGameButton.addActionListener(this);
   add(newGameButton);

   resignButton = new JButton("Resign");
   resignButton.addActionListener(this);
   add(resignButton);

   message = new JLabel("Click \"New Game\" to begin a game.");
   message.setForeground( new Color(100,255,100) ); // Light green.
   message.setFont(new Font("Serif", Font.BOLD, 14));
   add(message);

   /* Set the position and size of each component by calling
```

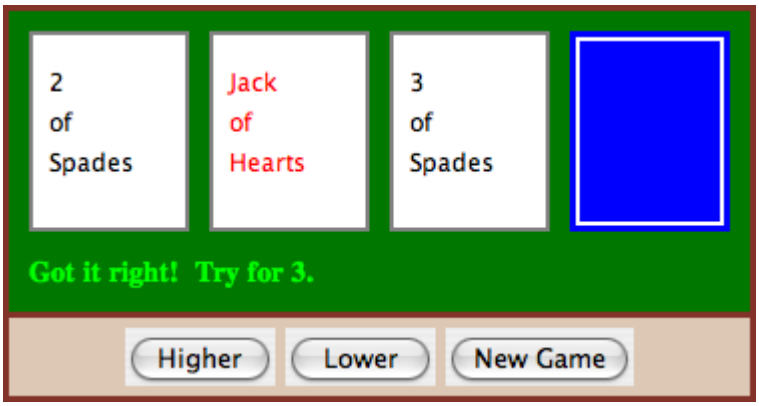
```
        its setBounds() method. */

      board.setBounds(20,20,164,164);
      newGameButton.setBounds(210, 60, 120, 30);
      resignButton.setBounds(210, 120, 120, 30);
      message.setBounds(20, 200, 330, 30);

   } // end constructor
```

It's reasonably easy, in this case, to get an attractive layout. It's much more difficult to do your own layout if you want to allow for changes of size. In that case, you have to respond to changes in the container's size by recomputing the sizes and positions of all the components that it contains. If you want to respond to changes in a container's size, you can register an appropriate listener with the container. Any component generates an event of type `ComponentEvent` when its size changes (and also when it is moved, hidden, or shown). You can register a `ComponentListener` with the container and respond to size change events by recomputing the sizes and positions of all the components in the container. Consult a Java reference for more information about `ComponentEvents`. However, my real advice is that if you want to allow for changes in the container's size, try to find a layout manager to do the work for you.

(The complete source code for this example is in *NullLayoutDemo.java*.)

6.7.6 A Little Card Game

For a final example, let's look at something a little more interesting as a program. The example is a simple card game in which you look at a playing card and try to predict whether the next card will be higher or lower in value. (Aces have the lowest value in this game.) You've seen a text-oriented version of the same game in Subsection 5.4.3. Section 5.4 also introduced *Deck*, *Hand*, and *Card* classes that are used in the game program. In this GUI version of the game, you click on a button to make your prediction. If you predict wrong, you lose. If you make three correct predictions, you win. After completing one game, you can click the "New Game" button to start a new game. Here is what the game looks like:



The game is implemented in a subclass of *JPanel* that is used as the content pane in the applet. The source code for the panel is *HighLowGUIPanel.java*. Applet and standalone versions of the program are defined by *HighLowGUIApplet.java* and *HighLowGUI.java*. You can try out the game in the on-line version of this section, or by running the program as a stand-alone application.

The overall structure of the main panel in this example should be clear: It has three buttons in a subpanel at the bottom of the main panel and a large drawing surface that displays the cards and a message. (The cards and message are not themselves components in this example; they are drawn in the panel's `paintComponent()` method.) The main panel uses a *BorderLayout*. The drawing surface occupies the `CENTER` position of the border layout. The subpanel that contains the buttons occupies the `SOUTH` position of the border layout, and the other three positions of the layout are empty.

The drawing surface is defined by a nested class named *CardPanel*, which is a subclass of *JPanel*. I have chosen to let the drawing surface object do most of the work of the game: It listens for events from the three buttons and responds by taking the appropriate actions. The main panel is defined by *HighLowGUIPanel* itself, which is another subclass of *JPanel*. The constructor of the *HighLowGUIPanel* class creates all the other components, sets up event handling, and lays out the components:

```
public HighLowGUIPanel() {   // The constructor.

   setBackground( new Color(130,50,40) );

   setLayout( new BorderLayout(3,3) );  // BorderLayout with 3-pixel gaps.

   CardPanel board = new CardPanel();  // Where the cards are drawn.
   add(board, BorderLayout.CENTER);

   JPanel buttonPanel = new JPanel();  // The subpanel that holds the buttons.
   buttonPanel.setBackground( new Color(220,200,180) );
   add(buttonPanel, BorderLayout.SOUTH);

   JButton higher = new JButton( "Higher" );
   higher.addActionListener(board);   // The CardPanel listens for events.
   buttonPanel.add(higher);

   JButton lower = new JButton( "Lower" );
   lower.addActionListener(board);
   buttonPanel.add(lower);

   JButton newGame = new JButton( "New Game" );
   newGame.addActionListener(board);
   buttonPanel.add(newGame);

   setBorder(BorderFactory.createLineBorder( new Color(130,50,40), 3) );

}  // end constructor
```

The programming of the drawing surface class, *CardPanel*, is a nice example of thinking in terms of a state machine. (See Subsection 6.5.4.) It is important to think in terms of the states that the game can be in, how the state can change, and how the response to events can depend on the state. The approach that produced the original, text-oriented game in Subsection 5.4.3 is not appropriate here. Trying to think about the game in terms of a process that goes step-by-step from beginning to end is more likely to confuse you than to help you.

The state of the game includes the cards and the message. The cards are stored in an object of type *Hand*. The message is a *String*. These values are stored in instance variables. There is also another, less obvious aspect of the state: Sometimes a game is in progress, and the user is supposed to make a prediction about the next card. Sometimes we are between games, and the user is supposed to click the "New Game" button. It's a good idea to keep

track of this basic difference in state. The *CardPanel* class uses a boolean instance variable named `gameInProgress` for this purpose.

The state of the game can change whenever the user clicks on a button. The *CardPanel* class implements the `ActionListener` interface and defines an `actionPerformed()` method to respond to the user's clicks. This method simply calls one of three other methods, `doHigher()`, `doLower()`, or `newGame()`, depending on which button was pressed. It's in these three event-handling methods that the action of the game takes place.

We don't want to let the user start a new game if a game is currently in progress. That would be cheating. So, the response in the `newGame()` method is different depending on whether the state variable `gameInProgress` is true or false. If a game is in progress, the `message` instance variable should be set to show an error message. If a game is not in progress, then all the state variables should be set to appropriate values for the beginning of a new game. In any case, the board must be repainted so that the user can see that the state has changed. The complete `newGame()` method is as follows:

```
/**
 * Called by the CardPanel constructor, and called by actionPerformed() if
 * the user clicks the "New Game" button.  Start a new game.
 */
void doNewGame() {
   if (gameInProgress) {
         // If the current game is not over, it is an error to try
         // to start a new game.
      message = "You still have to finish this game!";
      repaint();
      return;
   }
   deck = new Deck();   // Create the deck and hand to use for this game.
   hand = new Hand();
   deck.shuffle();
   hand.addCard( deck.dealCard() );  // Deal the first card into the hand.
   message = "Is the next card higher or lower?";
   gameInProgress = true;
   repaint();
} // end doNewGame()
```

The `doHigher()` and `doLower()` methods are almost identical to each other (and could probably have been combined into one method with a parameter, if I were more clever). Let's look at the `doHigher()` routine. This is called when the user clicks the "Higher" button. This only makes sense if a game is in progress, so the first thing `doHigher()` should do is check the value of the state variable `gameInProgress`. If the value is `false`, then `doHigher()` should just set up an error message. If a game is in progress, a new card should be added to the hand and the user's prediction should be tested. The user might win or lose at this time. If so, the value of the state variable `gameInProgress` must be set to `false` because the game is over. In any case, the board is repainted to show the new state. Here is the `doHigher()` method:

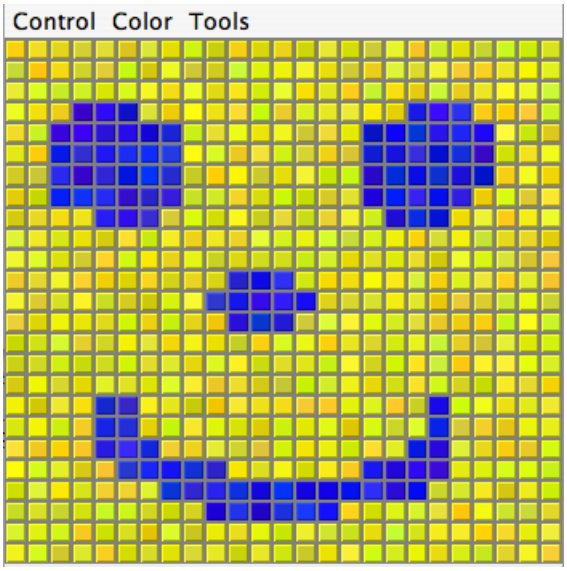
```
/**
 * Called by actionPerformed() when user clicks "Higher" button.
 * Check the user's prediction.  Game ends if user guessed
 * wrong or if the user has made three correct predictions.
 */
void doHigher() {
```

```
        if (gameInProgress == false) {
              // If the game has ended, it was an error to click "Higher",
              // So set up an error message and abort processing.
           message = "Click \"New Game\" to start a new game!";
           repaint();
           return;
        }
        hand.addCard( deck.dealCard() );     // Deal a card to the hand.
        int cardCt = hand.getCardCount();
        Card thisCard = hand.getCard( cardCt - 1 );  // Card just dealt.
        Card prevCard = hand.getCard( cardCt - 2 );  // The previous card.
        if ( thisCard.getValue() < prevCard.getValue() ) {
           gameInProgress = false;
           message = "Too bad! You lose.";
        }
        else if ( thisCard.getValue() == prevCard.getValue() ) {
           gameInProgress = false;
           message = "Too bad!  You lose on ties.";
        }
        else if ( cardCt == 4) {
           gameInProgress = false;
           message = "You win!  You made three correct guesses.";
        }
        else {
           message = "Got it right!  Try for " + cardCt + ".";
        }
        repaint();
     } // end doHigher()
```

The `paintComponent()` method of the *CardPanel* class uses the values in the state variables to decide what to show. It displays the string stored in the `message` variable. It draws each of the cards in the `hand`. There is one little tricky bit: If a game is in progress, it draws an extra face-down card, which is not in the hand, to represent the next card in the deck. Drawing the cards requires some care and computation. I wrote a method, "`void drawCard(Graphics g, Card card, int x, int y)`", which draws a card with its upper left corner at the point `(x,y)`. The `paintComponent()` routine decides where to draw each card and calls this routine to do the drawing. You can check out all the details in the source code, *HighLowGUIPanel.java*. (The playing cards used in this program are not very impressive. A version of the program with images that actually look like cards can be found in Subsection 13.1.3.)

6.8 Menus and Dialogs

We have already encountered many of the basic aspects of GUI programming, but professional programs use many additional features. We will cover some of the advanced features of Java GUI programming in Chapter 13, but in this section we look briefly at a few more basic features that are essential for writing GUI programs. I will discuss these features in the context of a "MosaicDraw" program that is shown in this picture:



As the user clicks-and-drags the mouse in the large drawing area of this program, it leaves a trail of little colored squares. There is some random variation in the color of the squares. (This is meant to make the picture look a little more like a real mosaic, which is a picture made out of small colored stones in which there would be some natural color variation.) There is a menu bar above the drawing area. The "Control" menu contains commands for filling and clearing the drawing area, along with a few options that affect the appearance of the picture. The "Color" menu lets the user select the color that will be used when the user draws. The "Tools" menu affects the behavior of the mouse. Using the default "Draw" tool, the mouse leaves a trail of single squares. Using the "Draw 3x3" tool, the mouse leaves a swath of colored squares that is three squares wide. There are also "Erase" tools, which let the user set squares back to their default black color.

The drawing area of the program is a panel that belongs to the *MosaicPanel* class, a subclass of *JPanel* that is defined in *MosaicPanel.java*. *MosaicPanel* is a highly reusable class for representing mosaics of colored rectangles. It does not directly support drawing on the mosaic, but it does support setting the color of each individual square. The MosaicDraw program installs a mouse listener on the panel; the mouse listener responds to mousePressed and mouseDragged events on the panel by setting the color of the square that contains the mouse. This is a nice example of applying a listener to an object to do something that was not programmed into the object itself.

Most of the programming for MosaicDraw can be found in *MosaicDrawController.java*. (It could have gone into the *MosaicPanel* class, if I had not decided to use that pre-existing class in unmodified form.) It is the *MosaicDrawController* class that creates a *MosaicPanel* object and adds a mouse listener to it. It also creates the menu bar that is shown at the top of the program and implements all the commands in the menu bar. It has an instance method `getMosaicPanel()` that returns a reference to the mosaic panel that it has created, and it has another instance method `getMenuBar()` that returns a menu bar for the program. These methods are used to obtain the panel and menu bar so that they can be added to an applet or a frame.

To get a working program, an object of type *JApplet* or *JFrame* is needed. The files *MosaicDrawApplet.java* and *MosaicDrawFrame.java* define the applet and frame versions of the program. These are rather simple classes; they simply create a *MosaicDrawController* object

and use its mosaic panel and menu bar. I urge you to study these files, along with *MosaicDrawController.java*. I will not be discussing all aspects of the code here, but you should be able to understand it all after reading this section. As for *MosaicPanel.java*, it uses some techniques that you would not understand at this point, but I encourage you to at least read the comments in this file to learn about the API for mosaic panels.

6.8.1 Menus and Menubars

MosaicDraw is the first example that we have seen that uses a menu bar. Fortunately, menus are very easy to use in Java. The items in a menu are represented by the class *JMenuItem* (this class and other menu-related classes are in package `javax.swing`). Menu items are used in almost exactly the same way as buttons. In fact, *JMenuItem* and *JButton* are both subclasses of a class, *AbstractButton*, that defines their common behavior. In particular, a *JMenuItem* is created using a constructor that specifies the text of the menu item, such as:

```
JMenuItem fillCommand = new JMenuItem("Fill");
```

You can add an *ActionListener* to a *JMenuItem* by calling the menu item's `addActionListener()` method. The `actionPerformed()` method of the action listener is called when the user selects the item from the menu. You can change the text of the item by calling its `setText(String)` method, and you can enable it and disable it using the `setEnabled(boolean)` method. All this works in exactly the same way as for a *JButton*.

The main difference between a menu item and a button, of course, is that a menu item is meant to appear in a menu rather than in a panel. A menu in Java is represented by the class *JMenu*. A *JMenu* has a name, which is specified in the constructor, and it has an `add(JMenuItem)` method that can be used to add a *JMenuItem* to the menu. So, the "Tools" menu in the MosaicDraw program could be created as follows, where `listener` is a variable of type `ActionListener`:

```
JMenu toolsMenu = new JMenu("Tools");  // Create a menu with name "Tools"

JMenuItem drawCommand = new JMenuItem("Draw");   // Create a menu item.
drawCommand.addActionListener(listener);         // Add listener to menu item.
toolsMenu.add(drawCommand);                      // Add menu item to menu.

JMenuItem eraseCommand = new JMenuItem("Erase"); // Create a menu item.
eraseCommand.addActionListener(listener);        // Add listener to menu item.
toolsMenu.add(eraseCommand);                     // Add menu item to menu.
   .
   .  // Create and add other menu items.
   .
```

Once a menu has been created, it must be added to a menu bar. A menu bar is represented by the class *JMenuBar*. A menu bar is just a container for menus. It does not have a name, and its constructor does not have any parameters. It has an `add(JMenu)` method that can be used to add menus to the menu bar. The name of the menu then appears in the menu bar. For example, the MosaicDraw program uses three menus, `controlMenu`, `colorMenu`, and `toolsMenu`. We could create a menu bar and add the menus to it with the statements:

```
JMenuBar menuBar = new JMenuBar();
menuBar.add(controlMenu);
menuBar.add(colorMenu);
menuBar.add(toolsMenu);
```

The final step in using menus is to use the menu bar in a *JApplet* or *JFrame*. We have already seen that an applet or frame has a "content pane." The menu bar is another component of the applet or frame, not contained inside the content pane. Both the *JApplet* and the *JFrame* classes include an instance method `setMenuBar(JMenuBar)` that can be used to set the menu bar. (There can only be one, so this is a "set" method rather than an "add" method.) In the MosaicDraw program, the menu bar is created by a *MosaicDrawController* object and can be obtained by calling that object's `getMenuBar()` method. Here is the basic code that is used (in somewhat modified form) to set up the interface both in the applet and in the frame version of the program:

```
MosaicDrawController controller = new MosaicDrawController();

MosaicPanel content = controller.getMosaicPanel();
setContentPane( content );  // Use panel from controller as content pane.

JMenuBar menuBar = controller.getMenuBar();
setJMenuBar( menuBar );    // Use the menu bar from the controller.
```

Using menus always follows the same general pattern: Create a menu bar. Create menus and add them to the menu bar. Create menu items and add them to the menus (and set up listening to handle action events from the menu items). Use the menu bar in a *JApplet* or *JFrame* by calling the `setJMenuBar()` method of the applet or frame.

* * *

There are other kinds of menu items, defined by subclasses of *JMenuItem*, that can be added to menus. One of these is *JCheckBoxMenuItem*, which represents menu items that can be in one of two states, selected or not selected. A *JCheckBoxMenuItem* has the same functionality and is used in the same way as a *JCheckBox* (see Subsection 6.6.3). Three *JCheckBoxMenuItems* are used in the "Control" menu of the MosaicDraw program. One can be used to turn the random color variation of the squares on and off. Another turns a symmetry feature on and off; when symmetry is turned on, the user's drawing is reflected horizontally and vertically to produce a symmetric pattern. And the third checkbox menu item shows and hides the "grouting" in the mosaic; the grouting is the gray lines that are drawn around each of the little squares in the mosaic. The menu item that corresponds to the "Use Randomness" option in the "Control" menu could be set up with the statements:

```
JMenuItem useRandomnessToggle = new JCheckBoxMenuItem("Use Randomness");
useRandomnessToggle.addActionListener(listener);  // Set up a listener.
useRandomnessToggle.setSelected(true);  // Randomness is initially turned on.
controlMenu.add(useRandomnessToggle);   // Add the menu item to the menu.
```

The "Use Randomness" *JCheckBoxMenuItem* corresponds to a boolean-valued instance variable named `useRandomness` in the *MosaicDrawController* class. This variable is part of the state of the controller object. Its value is tested whenever the user draws one of the squares, to decide whether or not to add a random variation to the color of the square. When the user selects the "Use Randomness" command from the menu, the state of the *JCheckBoxMenuItem* is reversed, from selected to not-selected or from not-selected to selected. The *ActionListener* for the menu item checks whether the menu item is selected or not, and it changes the value of `useRandomness` to match. Note that selecting the menu command does not have any immediate effect on the picture that is shown in the window. It just changes the state of the program so that future drawing operations on the part of the user will have a different effect. The "Use Symmetry" option in the "Control" menu works in much the same way. The "Show Grouting"

option is a little different. Selecting the "Show Grouting" option does have an immediate effect: The picture is redrawn with or without the grouting, depending on the state of the menu item.

My program uses a single *ActionListener* to respond to all of the menu items in all the menus. This is not a particularly good design, but it is easy to implement for a small program like this one. The `actionPerformed()` method of the listener object uses the statement

```
String command = evt.getActionCommand();
```

to get the action command of the source of the event; this will be the text of the menu item. The listener tests the value of `command` to determine which menu item was selected by the user. If the menu item is a *JCheckBoxMenuItem*, the listener must check the state of the menu item. The menu item is the source of the event that is being processed. The listener can get its hands on the menu item object by calling `evt.getSource()`. Since the return value of `getSource()` is of type *Object*, the return value must be type-cast to the correct type. Here, for example, is the code that handles the "Use Randomness" command:

```
if (command.equals("Use Randomness")) {
        // Set the value of useRandomness depending on the menu item's state.
   JCheckBoxMenuItem toggle = (JCheckBoxMenuItem)evt.getSource();
   useRandomness = toggle.isSelected();
}
```

(The `actionPerformed()` method uses a rather long `if..then..else` statement to check all the possible action commands. This would be a natural place to use a `switch` statement with `command` as the selector and all the possible action commands as cases. However, this can only be done if you are sure that the program will be run using Java 7 or later, since *Strings* were not allowed in `switch` statements in earlier versions of Java.)

* * *

In addition to menu items, a menu can contain lines that separate the menu items into groups. In the MosaicDraw program, the "Control" menu contains such a separator. A *JMenu* has an instance method `addSeparator()` that can be used to add a separator to the menu. For example, the separator in the "Control" menu was created with the statement:

```
controlMenu.addSeparator();
```

A menu can also contain a submenu. The name of the submenu appears as an item in the main menu. When the user moves the mouse over the submenu name, the submenu pops up. (There is no example of this in the MosaicDraw program.) It is very easy to do this in Java: You can add one *JMenu* to another *JMenu* using a statement such as `mainMenu.add(submenu)`.

6.8.2 Dialogs

One of the commands in the "Color" menu of the MosaicDraw program is "Custom Color...". When the user selects this command, a new window appears where the user can select a color. This window is an example of a ***dialog*** or ***dialog box***. A dialog is a type of window that is generally used for short, single purpose interactions with the user. For example, a dialog box can be used to display a message to the user, to ask the user a question, to let the user select a file to be opened, or to let the user select a color. In Swing, a dialog box is represented by an object belonging to the class *JDialog* or to a subclass.

The *JDialog* class is very similar to *JFrame* and is used in much the same way. Like a frame, a dialog box is a separate window. Unlike a frame, however, a dialog is not completely independent. Every dialog is associated with a frame (or another dialog), which is called

its ***parent window***. The dialog box is dependent on its parent. For example, if the parent is closed, the dialog box will also be closed. It is possible to create a dialog box without specifying a parent, but in that case an invisible frame is created by the system to serve as the parent.

Dialog boxes can be either ***modal*** or ***modeless***. When a modal dialog is created, its parent frame is blocked. That is, the user will not be able to interact with the parent until the dialog box is closed. Modeless dialog boxes do not block their parents in the same way, so they seem a lot more like independent windows. In practice, modal dialog boxes are easier to use and are much more common than modeless dialogs. All the examples we will look at are modal.

Aside from having a parent, a *JDialog* can be created and used in the same way as a *JFrame*. However, I will not give any examples here of using *JDialog* directly. Swing has many convenient methods for creating common types of dialog boxes. For example, the color choice dialog that appears when the user selects the "Custom Color" command in the MosaicDraw program belongs to the class *JColorChooser*, which is a subclass of *JDialog*. The *JColorChooser* class has a static method that makes color choice dialogs very easy to use:

```
Color JColorChooser.showDialog(Component parentComp,
                                        String title, Color initialColor)
```

When you call this method, a dialog box appears that allows the user to select a color. The first parameter specifies the parent of the dialog; the parent window of the dialog will be the window (if any) that contains `parentComp`; this parameter can be `null` and it can itself be a frame or dialog object. The second parameter is a string that appears in the title bar of the dialog box. And the third parameter, `initialColor`, specifies the color that is selected when the color choice dialog first appears. The dialog has a sophisticated interface that allows the user to change the selected color. When the user presses an "OK" button, the dialog box closes and the selected color is returned as the value of the method. The user can also click a "Cancel" button or close the dialog box in some other way; in that case, `null` is returned as the value of the method. This is a modal dialog, and the `showDialog()` does not return until the user dismisses the dialog box in some way. By using this predefined color chooser dialog, you can write one line of code that will let the user select an arbitrary color. Swing also has a *JFileChooser* class that makes it almost as easy to show a dialog box that lets the user select a file to be opened or saved.

The *JOptionPane* class includes a variety of methods for making simple dialog boxes that are variations on three basic types: a "message" dialog, a "confirm" dialog, and an "input" dialog. (The variations allow you to provide a title for the dialog box, to specify the icon that appears in the dialog, and to add other components to the dialog box. I will only cover the most basic forms here.) The on-line version of this section includes an applet that demonstrates *JOptionPane* as well as *JColorChooser*.

A message dialog simply displays a message string to the user. The user (hopefully) reads the message and dismisses the dialog by clicking the "OK" button. A message dialog can be shown by calling the static method:

```
void JOptionPane.showMessageDialog(Component parentComp, String message)
```

The message can be more than one line long. Lines in the message should be separated by newline characters, `\n`. New lines will not be inserted automatically, even if the message is very long.

An input dialog displays a question or request and lets the user type in a string as a response. You can show an input dialog by calling:

```
String JOptionPane.showInputDialog(Component parentComp, String question)
```

Again, the question can include newline characters. The dialog box will contain an input box, an "OK" button, and a "Cancel" button. If the user clicks "Cancel", or closes the dialog box in some other way, then the return value of the method is `null`. If the user clicks "OK", then the return value is the string that was entered by the user. Note that the return value can be an empty string (which is not the same as a `null` value), if the user clicks "OK" without typing anything in the input box. If you want to use an input dialog to get a numerical value from the user, you will have to convert the return value into a number; see Subsection 3.7.2.

Finally, a confirm dialog presents a question and three response buttons: "Yes", "No", and "Cancel". A confirm dialog can be shown by calling:

```
int JOptionPane.showConfirmDialog(Component parentComp, String question)
```

The return value tells you the user's response. It is one of the following constants:

- `JOptionPane.YES_OPTION` — the user clicked the "Yes" button
- `JOptionPane.NO_OPTION` — the user clicked the "No" button
- `JOptionPane.CANCEL_OPTION` — the user clicked the "Cancel" button
- `JOptionPane.CLOSE_OPTION` — the dialog was closed in some other way.

By the way, it is possible to omit the Cancel button from a confirm dialog by calling one of the other methods in the `JOptionPane` class. Just call:

```
JOptionPane.showConfirmDialog(
                parent, question, title, JOptionPane.YES_NO_OPTION )
```

The final parameter is a constant which specifies that only a "Yes" button and a "No" button should be used. The third parameter is a string that will be displayed as the title of the dialog box window.

If you would like to see how dialogs are created and used in the sample applet, you can find the source code in the file *SimpleDialogDemo.java*.

6.8.3 Fine Points of Frames

In previous sections, whenever I used a frame, I created a *JFrame* object in a `main()` routine and installed a panel as the content pane of that frame. This works fine, but a more object-oriented approach is to define a subclass of *JFrame* and to set up the contents of the frame in the constructor of that class. This is what I did in the case of the MosaicDraw program. *MosaicDrawFrame* is defined as a subclass of *JFrame*. The definition of this class is very short, but it illustrates several new features of frames that I want to discuss:

```
public class MosaicDrawFrame extends JFrame {

   public static void main(String[] args) {
      JFrame window = new MosaicDrawFrame();
      window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      window.setVisible(true);
   }

   public MosaicDrawFrame() {
      super("Mosaic Draw");
      MosaicDrawController controller = new MosaicDrawController();
      setContentPane( controller.getMosaicPanel() );
      setJMenuBar( controller.getMenuBar() );
      pack();
```

```
        Dimension screensize = Toolkit.getDefaultToolkit().getScreenSize();
        setLocation( (screensize.width - getWidth())/2,
                                    (screensize.height - getHeight())/2 );
    }

}
```

The constructor in this class begins with the statement `super("Mosaic Draw")`, which calls the constructor in the superclass, *JFrame*. The parameter specifies a title that will appear in the title bar of the window. The next three lines of the constructor set up the contents of the window; a *MosaicDrawController* is created, and the content pane and menu bar of the window are obtained from the controller. The next line is something new. If `window` is a variable of type *JFrame* (or *JDialog*), then the statement `window.pack()` will resize the window so that its size matches the preferred size of its contents. (In this case, of course, "`pack()`" is equivalent to "`this.pack()`"; that is, it refers to the window that is being created by the constructor.) The `pack()` method is usually the best way to set the size of a window. Note that it will only work correctly if every component in the window has a correct preferred size. This is only a problem in two cases: when a panel is used as a drawing surface and when a panel is used as a container with a `null` layout manager. In both these cases there is no way for the system to determine the correct preferred size automatically, and you should set a preferred size by hand. For example:

```
panel.setPreferredSize( new Dimension(400, 250) );
```

The last two lines in the constructor position the window so that it is exactly centered on the screen. The line

```
Dimension screensize = Toolkit.getDefaultToolkit().getScreenSize();
```

determines the size of the screen. The size of the screen is `screensize.width` pixels in the horizontal direction and `screensize.height` pixels in the vertical direction. The `setLocation()` method of the frame sets the position of the upper left corner of the frame on the screen. The expression "`screensize.width - getWidth()`" is the amount of horizontal space left on the screen after subtracting the width of the window. This is divided by 2 so that half of the empty space will be to the left of the window, leaving the other half of the space to the right of the window. Similarly, half of the extra vertical space is above the window, and half is below.

Note that the constructor has created the window and set its size and position, but that at the end of the constructor, the window is not yet visible on the screen. (More exactly, the constructor has created the window *object*, but the visual representation of that object on the screen has not yet been created.) To show the window on the screen, it will be necessary to call its instance method, `window.setVisible(true)`.

In addition to the constructor, the *MosaicDrawFrame* class includes a `main()` routine. This makes it possible to run *MosaicDrawFrame* as a stand-alone application. (The `main()` routine, as a `static` method, has nothing to do with the function of a *MosaicDrawFrame* object, and it could (and perhaps should) be in a separate class.) The `main()` routine creates a *MosaicDrawFrame* and makes it visible on the screen. It also calls

```
window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

which means that the program will end when the user closes the window. Note that this is not done in the constructor because doing it there would make *MosaicDrawFrame* less flexible. It is possible, for example, to write a program that lets the user open multiple MosaicDraw windows. In that case, we don't want to end the program just because the user has closed *one*

of the windows. Furthermore, it is possible for an applet to create a frame, which will open as a separate window on the screen. An applet is not allowed to "terminate the program" (and it's not even clear what that should mean in the case of an applet), and attempting to do so will produce an exception. There are other possible values for the default close operation of a window:

- `JFrame.DO_NOTHING_ON_CLOSE` — the user's attempts to close the window by clicking its close box will be ignored.
- `JFrame.HIDE_ON_CLOSE` — when the user clicks its close box, the window will be hidden just as if `window.setVisible(false)` were called. The window can be made visible again by calling `window.setVisible(true)`. This is the value that is used if you do not specify another value by calling `setDefaultCloseOperation`.
- `JFrame.DISPOSE_ON_CLOSE` — the window is closed and any operating system resources used by the window are released. It is not possible to make the window visible again. (This is the proper way to permanently get rid of a window without ending the program. You can accomplish the same thing by calling the instance method `window.dispose()`.)

I've written an applet version of the MosaicDraw program that appears on a Web page as a single button. When the user clicks the button, the applet opens a *MosaicDrawFrame*. In this case, the applet sets the default close operation of the window to `JFrame.DISPOSE_ON_CLOSE`. You can try the applet in the on-line version of this section.

The file *MosaicDrawLauncherApplet.java* contains the source code for the applet. One interesting point in the applet is that the text of the button changes depending on whether a window is open or not. If there is no window, the text reads "Launch MosaicDraw". When the window is open, it changes to "Close MosaicDraw", and clicking the button will close the window. The change is implemented by attaching a *WindowListener* to the window. The listener responds to *WindowEvents* that are generated when the window opens and closes. Although I will not discuss window events further here, you can look at the source code for an example of how they can be used.

6.8.4 Creating Jar Files

As the final topic for this chapter, we look again at jar files. Recall that a jar file is a "java archive" that can contain a number of class files. When creating a program that uses more than one class, it's usually a good idea to place all the classes that are required by the program into a jar file. If that is done, then a user will only need that one file to run the program. Subsection 6.2.4 discusses how a jar file can be used for an applet. Jar files can also be used for stand-alone applications. In fact, it is possible to make a so-called ***executable jar file***. A user can run an executable jar file in much the same way as any other application, usually by double-clicking the icon of the jar file. (The user's computer must have a correct version of Java installed, and the computer must be configured correctly for this to work. The configuration is usually done automatically when Java is installed, at least on Windows and Mac OS.)

The question, then, is how to create a jar file. The answer depends on what programming environment you are using. The two basic types of programming environment—command line and IDE—were discussed in Section 2.6. Any IDE (Integrated Programming Environment) for Java should have a command for creating jar files. In the Eclipse IDE, for example, it can be done as follows: In the Package Explorer pane, select the programming project (or just all the individual source code files that you need). Right-click on the selection, and choose "Export"

from the menu that pops up. In the window that appears, select "JAR file" and click "Next". In the window that appears next, enter a name for the jar file in the box labeled "JAR file". (Click the "Browse" button next to this box to select the file name using a file dialog box.) The name of the file should end with ".jar". If you are creating a regular jar file, not an executable one, you can hit "Finish" at this point, and the jar file will be created. You could do this, for example, if the jar file contains an applet but no main program. To create an executable file, hit the "Next" button *twice* to get to the "Jar Manifest Specification" screen. At the bottom of this screen is an input box labeled "Main class". You have to enter the name of the class that contains the `main()` routine that will be run when the jar file is executed. If you hit the "Browse" button next to the "Main class" box, you can select the class from a list of classes that contain `main()` routines. Once you've selected the main class, you can click the "Finish" button to create the executable jar file. (Note that newer versions of Eclipse also have an option for exporting an executable Jar file in fewer steps.)

It is also possible to create jar files on the command line. The Java Development Kit includes a command-line program named `jar` that can be used to create jar files. If all your classes are in the default package (like most of the examples in this book), then the `jar` command is easy to use. To create a non-executable jar file on the command line, change to the directory that contains the class files that you want to include in the jar. Then give the command

```
jar  cf  JarFileName.jar  *.class
```

where `JarFileName` can be any name that you want to use for the jar file. The "`*`" in "`*.class`" is a wildcard that makes `*.class` match every class file in the current directory. This means that all the class files in the directory will be included in the jar file. If you want to include only certain class files, you can name them individually, separated by spaces. (Things get more complicated if your classes are not in the default package. In that case, the class files must be in subdirectories of the directory in which you issue the `jar` command. See Subsection 2.6.4.)

Making an executable jar file on the command line is more complicated. There has to be some way of specifying which class contains the `main()` routine. This is done by creating a ***manifest file***. The manifest file can be a plain text file containing a single line of the form

```
Main-Class: ClassName
```

where `ClassName` should be replaced by the name of the class that contains the `main()` routine. For example, if the `main()` routine is in the class *MosaicDrawFrame*, then the manifest file should read "`Main-Class:  MosaicDrawFrame`". You can give the manifest file any name you like. Put it in the same directory where you will issue the `jar` command, and use a command of the form

```
jar  cmf  ManifestFileName  JarFileName.jar  *.class
```

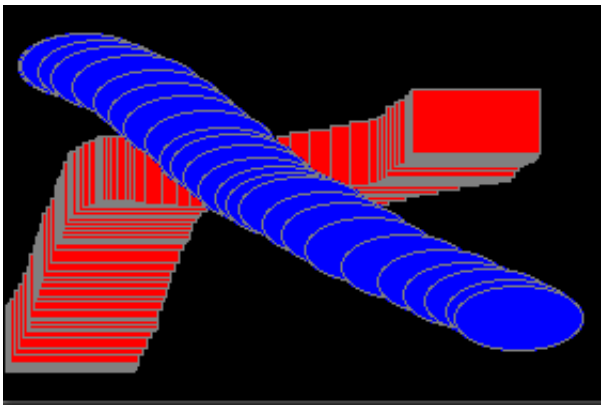
to create the jar file. (The `jar` command is capable of performing a variety of different operations. The first parameter to the command, such as "`cf`" or "`cmf`", tells it which operation to perform.)

By the way, if you have successfully created an executable jar file, you can run it on the command line using the command "`java -jar`". For example:

```
java  -jar  JarFileName.jar
```

Exercises for Chapter 6

1. In the *SimpleStamperPanel* example from Subsection 6.4.2, a rectangle or oval is drawn on the panel when the user clicks the mouse, except that when the user shift-clicks, the panel is cleared instead. Modify this class so that the modified version will continue to draw figures as the user drags the mouse. That is, the mouse will leave a trail of figures as the user drags. However, if the user shift-clicks, the panel should simply be cleared and no figures should be drawn even if the user drags the mouse after shift-clicking. Use your panel either in an applet or in a stand-alone application (or both). Here is a picture of my solution:



The source code for the original panel class is *SimpleStamperPanel.java*. An applet that uses this class can be found in *SimpleStamperApplet.java*, and a main program that uses the panel in a frame is in *SimpleStamper.java*. See the discussion of dragging in Subsection 6.4.4. (Note that in the original version, I drew a black outline around each shape. In the modified version, I decided that it would look better to draw a gray outline instead.)

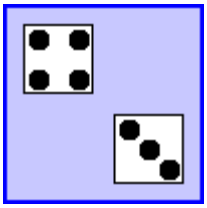
If you want to make the problem a little more challenging, when drawing shapes during a drag operation, make sure that the shapes that are drawn are at least, say, 5 pixels apart. To implement this, you have to keep track of the position of the last shape that was drawn.

2. Write a panel that shows a small red square and a small blue square. The user should be able to drag either square with the mouse. (You'll need an instance variable to remember which square the user is dragging.) The user can drag the square off the applet if she wants; if she does this, there is no way to get it back. Use your panel in either an applet or a stand-alone application.

Note that for this exercise, you should do all the drawing in the `paintComponent()` method (as indeed you should whenever possible).

3. Write a panel that shows a pair of dice. When the user clicks on the panel, the dice should be rolled (that is, the dice should be assigned newly computed random values). Each die should be drawn as a square showing from 1 to 6 dots. Since you have to draw two dice, its a good idea to write a subroutine, "`void drawDie(Graphics g, int val, int x, int y)`", to draw a die at the specified `(x,y)` coordinates. The second parameter, `val`, specifies the value that is showing on the die. Assume that the size of the panel is 100

by 100 pixels. Also write an applet that uses your panel as its content pane. Here is a picture of the applet:



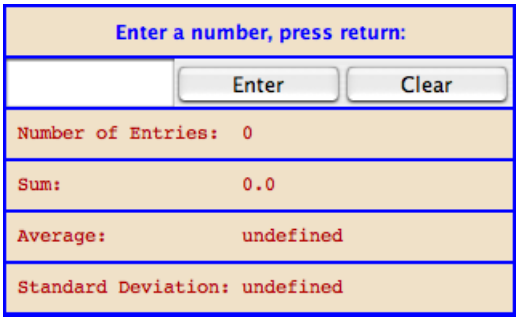
4. In Exercise 6.3, you wrote a pair-of-dice panel where the dice are rolled when the user clicks on the panel. Now make a pair-of-dice program in which the user rolls the dice by clicking a button. The button should appear under the panel that shows the dice. Also make the following change: When the dice are rolled, instead of just showing the new value, show a short animation during which the values on the dice are changed in every frame. The animation is supposed to make the dice look more like they are actually rolling. Write your program as a stand-alone application.

5. In Exercise 3.6, you drew a checkerboard. For this exercise, write a checkerboard applet where the user can select a square by clicking on it. Highlight the selected square by drawing a colored border around it. When the applet is first created, no square is selected. When the user clicks on a square that is not currently selected, it becomes selected (and the previously selected square, if any, is unselected). If the user clicks the square that is selected, it becomes unselected. Assume that the size of the applet is exactly 160 by 160 pixels, so that each square on the checkerboard is 20 by 20 pixels.

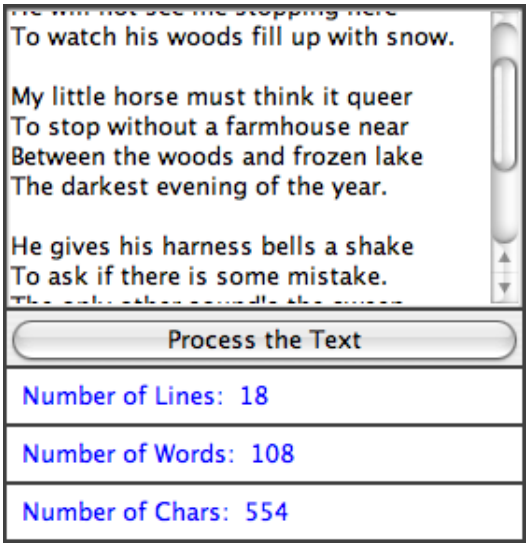
6. For this exercise, you should modify the SubKiller game from Subsection 6.5.4. You can start with the existing source code, from the file *SubKillerPanel.java*. Modify the game so it keeps track of the number of hits and misses and displays these quantities. That is, every time the depth charge blows up the sub, the number of hits goes up by one. Every time the depth charge falls off the bottom of the screen without hitting the sub, the number of misses goes up by one. There is room at the top of the panel to display these numbers. To do this exercise, you only have to add a half-dozen lines to the source code. But you have to figure out what they are and where to add them. To do this, you'll have to read the source code closely enough to understand how it works.

7. Exercise 5.2 involved a class, *StatCalc.java*, that could compute some statistics of a set of numbers. Write a program that uses the *StatCalc* class to compute and display statistics of numbers entered by the user. The panel will have an instance variable of type *StatCalc* that does the computations. The panel should include a *JTextField* where the user enters a number. It should have four labels that display four statistics for the numbers that have been entered: the number of numbers, the sum, the mean, and the standard deviation. Every time the user enters a new number, the statistics displayed on the labels should change. The user enters a number by typing it into the *JTextField* and pressing return. There should be a "Clear" button that clears out all the data. This means creating a new *StatCalc* object and resetting the displays on the labels. My panel also has an "Enter" button that does the same thing as pressing the return key in the *JTextField*. (Recall that a *JTextField* generates an *ActionEvent* when the user presses return, so your panel should

register itself to listen for *ActionEvents* from the *JTextField*.) Write your program as a stand-alone application. Here is a picture of my solution to this problem:



8. Write a panel with a *JTextArea* where the user can enter some text. The panel should have a button. When the user clicks on the button, the panel should count the number of lines in the user's input, the number of words in the user's input, and the number of characters in the user's input. This information should be displayed on three labels in the panel. Recall that if `textInput` is a *JTextArea*, then you can get the contents of the *JTextArea* by calling the function `textInput.getText()`. This function returns a *String* containing all the text from the text area. The number of characters is just the length of this *String*. Lines in the *String* are separated by the new line character, `'\n'`, so the number of lines is just the number of new line characters in the *String*, plus one. Words are a little harder to count. Exercise 3.4 has some advice about finding the words in a *String*. Essentially, you want to count the number of characters that are first characters in words. Don't forget to put your *JTextArea* in a *JScrollPane*, and add the scroll pane to the container, not the text area. Scrollbars should appear when the user types more text than will fit in the available area. Here is a picture of my solution:



9. Write a GUI Blackjack program that lets the user play a game of Blackjack, with the computer as the dealer. The applet should draw the user's cards and the dealer's cards,

just as was done for the graphical HighLow card game in Subsection 6.7.6. You can use the source code for that game, *HighLowGUI.java*, for some ideas about how to write your Blackjack game. The structures of the HighLow panel and the Blackjack panel are very similar. You will certainly want to use the `drawCard()` method from the HighLow program.

You can find a description of the game of Blackjack in Exercise 5.5. Add the following rule to that description: If a player takes five cards without going over 21, that player wins immediately. This rule is used in some casinos. For your program, it means that you only have to allow room for five cards. You should assume that the panel is just wide enough to show five cards, and that it is tall enough show the user's hand and the dealer's hand.

Note that the design of a GUI Blackjack game is very different from the design of the text-oriented program that you wrote for Exercise 5.5. The user should play the game by clicking on "Hit" and "Stand" buttons. There should be a "New Game" button that can be used to start another game after one game ends. You have to decide what happens when each of these buttons is pressed. You don't have much chance of getting this right unless you think in terms of the states that the game can be in and how the state can change.

Your program will need the classes defined in *Card.java*, *Hand.java*, *Deck.java*, and *BlackjackHand.java*.

10. In the Blackjack game from Exercise 6.9, the user can click on the "Hit", "Stand", and "NewGame" buttons even when it doesn't make sense to do so. It would be better if the buttons were disabled at the appropriate times. The "New Game" button should be disabled when there is a game in progress. The "Hit" and "Stand" buttons should be disabled when there is not a game in progress. The instance variable `gameInProgress` tells whether or not a game is in progress, so you just have to make sure that the buttons are properly enabled and disabled whenever this variable changes value. I strongly advise writing a subroutine that can be called whenever it is necessary to set the value of the `gameInProgress` variable. Then the subroutine can take responsibility for enabling and disabling the buttons. Recall that if `bttn` is a variable of type `JButton`, then `bttn.setEnabled(false)` disables the button and `bttn.setEnabled(true)` enables the button.

As a second (and more difficult) improvement, make it possible for the user to place bets on the Blackjack game. When the applet starts, give the user $100. Add a *JTextField* to the strip of controls along the bottom of the applet. The user can enter the bet in this *JTextField*. When the game begins, check the amount of the bet. You should do this when the game begins, not when it ends, because several errors can occur: The contents of the *JTextField* might not be a legal number. The bet that the user places might be more money than the user has, or it might be <= 0. You should detect these errors and show an error message instead of starting the game. The user's bet should be an integral number of dollars.

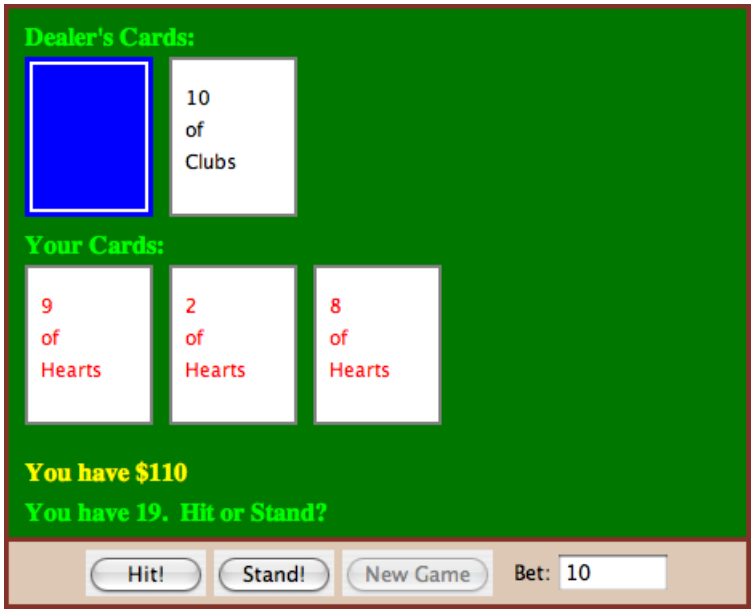
It would be a good idea to make the *JTextField* uneditable while the game is in progress. If `betInput` is the *JTextField*, you can make it editable and uneditable by the user with the commands `betInput.setEditable(true)` and `betInput.setEditable(false)`.

In the `paintComponent()` method, you should include commands to display the amount of money that the user has left.

There is one other thing to think about: Ideally, the applet should not start a new game when it is first created. The user should have a chance to set a bet amount before the game starts. So, in the constructor for the drawing surface class, you should not call

`doNewGame()`. You might want to display a message such as "Welcome to Blackjack" before the first game starts.

Here is a picture of my program:



Quiz on Chapter 6

1. Programs written for a graphical user interface have to deal with "events." Explain what is meant by the term *event*. Give at least two different examples of events, and discuss how a program might respond to those events.

2. Explain carefully what the `repaint()` method does.

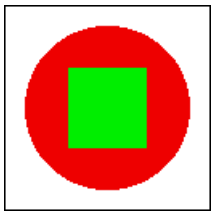
3. What is HTML?

4. Java has a standard class called *JPanel*. Discuss **two** ways in which JPanels can be used.

5. Draw the picture that will be produced by the following `paintComponent()` method:

```
public static void paintComponent(Graphics g) {
    super.paintComponent(g);
    for (int i=10; i <= 210; i = i + 50)
       for (int j = 10; j <= 210; j = j + 50)
          g.drawLine(i,10,j,60);
}
```

6. Suppose you would like a panel that displays a green square inside a red circle, as illustrated. Write a `paintComponent()` method for the panel class that will draw the image.

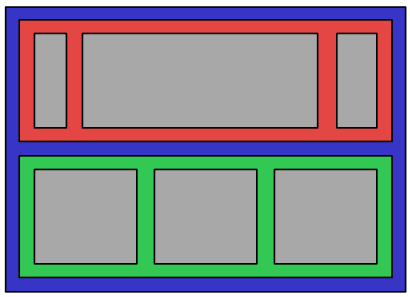


7. Java has a standard class called *MouseEvent*. What is the purpose of this class? What does an object of type *MouseEvent* do?

8. One of the main classes in Swing is the *JComponent* class. What is meant by a component? What are some examples?

9. What is the function of a *LayoutManager* in Java?

10. What type of layout manager is being used for each of the three panels in the following illustration from Section 6.7?



Three panels, shown in color, containing six other components, shown in gray.

11. Explain how *Timers* are used to do animation.

12. What is a *JCheckBox* and how is it used?

Chapter 7

Arrays

COMPUTERS GET A LOT OF THEIR POWER from working with ***data structures***. A data structure is an organized collection of related data. An object is a data structure, but this type of data structure—consisting of a fairly small number of named instance variables—is just the beginning. In many cases, programmers build complicated data structures by hand, by linking objects together. We'll look at these custom-built data structures in Chapter 9. But there is one type of data structure that is so important and so basic that it is built into every programming language: the array.

An ***array*** is a data structure consisting of a numbered list of items, where all the items are of the same type. In Java, the items in an array are always numbered from zero up to some maximum value, which is set when the array is created. For example, an array might contain 100 integers, numbered from zero to 99. The items in an array can belong to one of Java's primitive types. They can also be references to objects, so that you could, for example, make an array containing all the buttons in a GUI program.

This chapter discusses how arrays are created and used in Java. It also covers the standard class `java.util.ArrayList`. An object of type *ArrayList* is very similar to an array of *Objects*, but it can grow to hold any number of items.

7.1 Creating and Using Arrays

WHEN A NUMBER OF DATA ITEMS are chunked together into a unit, the result is a ***data structure***. Data structures can be very complex, but in many applications, the appropriate data structure consists simply of a sequence of data items. Data structures of this simple variety can be either ***arrays*** or ***records***.

The term "record" is not used in Java. A record is essentially the same as a Java object that has instance variables only, but no instance methods. Some other languages, which do not support objects in general, nevertheless do support records. The C programming language, for example, is not object-oriented, but it has records, which in C go by the name "struct." The data items in a record—in Java, an object's instance variables—are called the ***fields*** of the record. Each item is referred to using a ***field name***. In Java, field names are just the names of the instance variables. The distinguishing characteristics of a record are that the data items in the record are referred to by name and that different fields in a record are allowed to be of different types. For example, if the class *Person* is defined as:

```
class Person {
   String name;
```

```
    int id_number;
    Date birthday;
    int age;
}
```

then an object of class *Person* could be considered to be a record with four fields. The field names are `name`, `id_number`, `birthday`, and `age`. Note that the fields are of various types: *String*, **int**, and *Date*.

Because records are just a special type of object, I will not discuss them further.

7.1.1 Arrays

Like a record, an array is a sequence of items. However, where items in a record are referred to by **name**, the items in an array are numbered, and individual items are referred to by their **position number**. Furthermore, all the items in an array must be of the same type. The definition of an array is: a numbered sequence of items, which are all of the same type. The number of items in an array is called the ***length*** of the array. The position number of an item in an array is called the ***index*** of that item. The type of the individual items in an array is called the ***base type*** of the array.

The base type of an array can be any Java type, that is, one of the primitive types, or a class name, or an interface name. If the base type of an array is **int**, it is referred to as an "array of **ints**." An array with base type *String* is referred to as an "array of *Strings*." However, an array is not, properly speaking, a list of integers or strings or other **values**. It is better thought of as a list of **variables** of type **int**, or a list of variables of type *String*, or of some other type. As always, there is some potential for confusion between the two uses of a variable: as a name for a memory location and as a name for the value stored in that memory location. Each position in an array acts as a variable. Each position can hold a value of a specified type (the base type of the array). The value can be changed at any time. Values are stored **in** an array. The array **is** the container, not the values.

The items in an array—really, the individual variables that make up the array—are more often referred to as the ***elements*** of the array. In Java, the elements in an array are always numbered starting from zero. That is, the index of the first element in the array is zero. If the length of the array is `N`, then the index of the last element in the array is `N-1`. Once an array has been created, its length cannot be changed.

Java arrays are **objects**. This has several consequences. Arrays are created using a special form of the `new` operator. No variable can ever **hold** an array; a variable can only **refer** to an array. Any variable that can refer to an array can also hold the value `null`, meaning that it doesn't at the moment refer to anything. Like any object, an array belongs to a class, which like all classes is a subclass of the class *Object*. The elements of the array are, essentially, instance variables in the array object, except that they are referred to by number rather than by name.

Nevertheless, even though arrays are objects, there are differences between arrays and other kinds of objects, and there are a number of special language features in Java for creating and using arrays.

7.1.2 Using Arrays

Suppose that `A` is a variable that refers to an array. Then the element at index `k` in `A` is referred to as `A[k]`. The first element is `A[0]`, the second is `A[1]`, and so forth. "`A[k]`" is really a variable, and it can be used just like any other variable. You can assign values to it, you can

use it in expressions, and you can pass it as a parameter to a subroutine. All of this will be discussed in more detail below. For now, just keep in mind the syntax

⟨*array-variable*⟩ [⟨*integer-expression*⟩]

for referring to an element of an array.

Although every array, as an object, belongs to some class, array classes never have to be defined. Once a type exists, the corresponding array class exists automatically. If the name of the type is *BaseType*, then the name of the associated array class is BaseType[]. That is to say, an object belonging to the class BaseType[] is an array of items, where each item is a variable of type *BaseType*. The brackets, "[]", are meant to recall the syntax for referring to the individual items in the array. "BaseType[]" is read as "array of BaseType" or "BaseType array." It might be worth mentioning here that if *ClassA* is a subclass of *ClassB*, then the class ClassA[] is automatically a subclass of ClassB[].

The base type of an array can be any legal Java type. From the primitive type **int**, the array type int[] is derived. Each element in an array of type int[] is a variable of type **int**, which holds a value of type **int**. From a class named *Shape*, the array type Shape[] is derived. Each item in an array of type Shape[] is a variable of type *Shape*, which holds a value of type *Shape*. This value can be either `null` or a reference to an object belonging to the class *Shape*. (This includes objects belonging to subclasses of *Shape*.)

* * *

Let's try to get a little more concrete about all this, using arrays of integers as our first example. Since int[] is a class, it can be used to declare variables. For example,

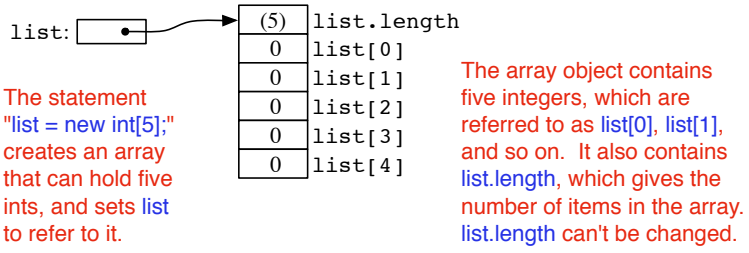
```
int[] list;
```

creates a variable named `list` of type int[]. This variable is **capable** of referring to an array of `int`s, but initially its value is `null` (if `list` is a member variable in a class) or undefined (if `list` is a local variable in a method). The `new` operator is used to create a new array object, which can then be assigned to `list`. The syntax for using `new` with arrays is different from the syntax you learned previously. As an example,

```
list = new int[5];
```

creates an array of five integers. More generally, the constructor "`new BaseType[N]`" is used to create an array belonging to the class BaseType[]. The value `N` in brackets specifies the length of the array, that is, the number of elements that it contains. Note that the array "knows" how long it is. The length of the array is an instance variable in the array object. In fact, the length of an array, `list`, can be referred to as `list.length`. (However, you are not allowed to change the value of `list.length`, so it's really a "`final`" instance variable, that is, one whose value cannot be changed after it has been initialized.)

The situation produced by the statement "`list = new int[5];`" can be pictured like this:



Note that the newly created array of integers is automatically filled with zeros. In Java, a newly created array is **always** filled with a known, default value: zero for numbers, `false` for **boolean**, the character with Unicode number zero for **char**, and `null` for objects.

The elements in the array, `list`, are referred to as `list[0]`, `list[1]`, `list[2]`, `list[3]`, and `list[4]`. (Note again that the index for the last item is one less than `list.length`.) However, array references can be much more general than this. The brackets in an array reference can contain any expression whose value is an integer. For example if `indx` is a variable of type **int**, then `list[indx]` and `list[2*indx+7]` are syntactically correct references to elements of the array `list`. Thus, the following loop would print all the integers in the array, `list`, to standard output:

```
for (int i = 0; i < list.length; i++) {
   System.out.println( list[i] );
}
```

The first time through the loop, `i` is 0, and `list[i]` refers to `list[0]`. So, it is the value stored in the variable `list[0]` that is printed. The second time through the loop, `i` is 1, and the value stored in `list[1]` is printed. The loop ends after printing the value of `list[4]`, when `i` becomes equal to 5 and the continuation condition "`i < list.length`" is no longer true. This is a typical example of using a loop to process an array. I'll discuss more examples of array processing throughout this chapter.

Every use of a variable in a program specifies a memory location. Think for a moment about what the computer does when it encounters a reference to an array element, `list[k]`, while it is executing a program. The computer must determine which memory location is being referred to. To the computer, `list[k]` means something like this: "Get the pointer that is stored in the variable, `list`. Follow this pointer to find an array object. Get the value of `k`. Go to the `k`-th position in the array, and that's the memory location you want." There are two things that can go wrong here. Suppose that the value of `list` is `null`. If that is the case, then `list` doesn't even refer to an array. The attempt to refer to an element of an array that doesn't exist is an error that will cause an exception of type *NullPointerException* to be thrown. The second possible error occurs if `list` does refer to an array, but the value of `k` is outside the legal range of indices for that array. This will happen if `k < 0` or if `k >= list.length`. This is called an "array index out of bounds" error. When an error of this type occurs, an exception of type *ArrayIndexOutOfBoundsException* is thrown. When you use arrays in a program, you should be mindful that both types of error are possible. However, array index out of bounds errors are by far the most common error when working with arrays.

7.1.3 Array Initialization

For an array variable, just as for any variable, you can declare the variable and initialize it in a single step. For example,

```
int[] list = new int[5];
```

If `list` is a local variable in a subroutine, then this is exactly equivalent to the two statements:

```
int[] list;
list = new int[5];
```

(If `list` is an instance variable, then of course you can't simply replace "`int[] list = new int[5];`" with "`int[] list; list = new int[5];`" since the assignment statement "`list = new int[5];`" is only legal inside a subroutine.)

The new array is filled with the default value appropriate for the base type of the array—zero for **int** and `null` for class types, for example. However, Java also provides a way to initialize an array variable with a new array filled with a specified list of values. In a declaration statement that creates a new array, this is done with an ***array initializer***. For example,

```
int[] list = { 1, 4, 9, 16, 25, 36, 49 };
```

creates a new array containing the seven values 1, 4, 9, 16, 25, 36, and 49, and sets `list` to refer to that new array. The value of `list[0]` will be 1, the value of `list[1]` will be 4, and so forth. The length of `list` is seven, since seven values are provided in the initializer.

An array initializer takes the form of a list of values, separated by commas and enclosed between braces. The length of the array does not have to be specified, because it is implicit in the list of values. The items in an array initializer don't have to be constants. They can be variables or arbitrary expressions, provided that their values are of the appropriate type. For example, the following declaration creates an array of eight *Colors*. Some of the colors are given by expressions of the form "`new Color(r,g,b)`" instead of by constants:

```
Color[] palette = {
             Color.BLACK,
             Color.RED,
             Color.PINK,
             new Color(0,180,0),  // dark green
             Color.GREEN,
             Color.BLUE,
             new Color(180,180,255),  // light blue
             Color.WHITE
           };
```

A list initializer of this form can be used **only** in a declaration statement, to give an initial value to a newly declared array variable. It cannot be used in an assignment statement to assign a value to a variable that has been previously declared. However, there is another, similar notation for creating a new array that can be used in an assignment statement or passed as a parameter to a subroutine. The notation uses another form of the `new` operator to both create and initialize a new array object at the same time. (The rather odd syntax is similar to the syntax for anonymous classes, which were discussed in Subsection 5.7.3.) For example to assign a new value to an array variable, `list`, that was declared previously, you could use:

```
list = new int[] { 1, 8, 27, 64, 125, 216, 343 };
```

The general syntax for this form of the `new` operator is

```
new ⟨base-type⟩ [ ] { ⟨list-of-values⟩ }
```

This is actually an expression whose value is a reference to a newly created array object. This means that it can be used in any context where an object of type ⟨*base-type*⟩`[]` is expected. For example, if `makeButtons` is a method that takes an array of *Strings* as a parameter, you could say:

```
makeButtons( new String[] { "Stop", "Go", "Next", "Previous" } );
```

Being able to create and use an array "in place" in this way can be very convenient, in the same way that anonymous nested classes are convenient.

By the way, it is perfectly legal to use the "`new BaseType[] { ... }`" syntax instead of the array initializer syntax in the declaration of an array variable. For example, instead of saying:

```
int[] primes = { 2, 3, 5, 7, 11, 13, 17, 19 };
```

you can say, equivalently,

```
int[] primes = new int[] { 2, 3, 5, 7, 11, 17, 19 };
```

In fact, rather than use a special notation that works only in the context of declaration statements, I prefer to use the second form.

* * *

One final note: For historical reasons, an array declaration such as

```
int[] list;
```

can also be written as

```
int list[];
```

which is a syntax used in the languages C and C++. However, this alternative syntax does not really make much sense in the context of Java, and it is probably best avoided. After all, the intent is to declare a variable of a certain type, and the name of that type is "int[]". It makes sense to follow the "⟨*type-name*⟩ ⟨*variable-name*⟩;" syntax for such declarations.

7.2 Programming With Arrays

Arrays are the most basic and the most important type of data structure, and techniques for processing arrays are among the most important programming techniques you can learn. Two fundamental array processing techniques—searching and sorting—will be covered in Section 7.4. This section introduces some of the basic ideas of array processing in general.

7.2.1 Arrays and for Loops

In many cases, processing an array means applying the same operation to each item in the array. This is commonly done with a `for` loop. A loop for processing all the elements of an array `A` has the form:

```
// do any necessary initialization
for (int i = 0; i < A.length; i++) {
   . . . // process A[i]
}
```

Suppose, for example, that `A` is an array of type double[]. Suppose that the goal is to add up all the numbers in the array. An informal algorithm for doing this would be:

```
Start with sum = 0;
Add A[0] to sum;   (process the first item in A)
Add A[1] to sum;   (process the second item in A)
  .
  .
  .
Add A[ A.length - 1 ] to sum;   (process the last item in A)
```

Putting the obvious repetition into a loop, this becomes:

```
double sum;  // The sum of the numbers in A.
sum = 0;     // Start with 0.
for (int i = 0; i < A.length; i++)
   sum += A[i];  // add A[i] to the sum, for
                 //     i = 0, 1, ..., A.length - 1
```

Note that the continuation condition, "`i < A.length`", implies that the last value of `i` that is actually processed is `A.length-1`, which is the index of the final item in the array. It's important to use "`<`" here, not "`<=`", since "`<=`" would give an array index out of bounds error. There is no element at position `A.length` in `A`.

Eventually, you should just about be able to write loops similar to this one in your sleep. I will give a few more simple examples. Here is a loop that will count the number of items in the array `A` which are less than zero:

```
int count;  // For counting the items.
count = 0;  // Start with 0 items counted.
for (int i = 0; i < A.length; i++) {
   if (A[i] < 0.0)   // if this item is less than zero...
      count++;       //     ...then count it
}
// At this point, the value of count is the number
// of items that have passed the test of being < 0
```

Replace the test "`A[i] < 0.0`", if you want to count the number of items in an array that satisfy some other property. Here is a variation on the same theme. Suppose you want to count the number of times that an item in the array `A` is equal to the item that follows it. The item that follows `A[i]` in the array is `A[i+1]`, so the test in this case is "`if (A[i] == A[i+1])`". But there is a catch: This test cannot be applied when `A[i]` is the last item in the array, since then there is no such item as `A[i+1]`. The result of trying to apply the test in this case would be an *ArrayIndexOutOfBoundsException*. This just means that we have to stop one item short of the final item:

```
int count = 0;
for (int i = 0; i < A.length - 1; i++) {
   if (A[i] == A[i+1])
      count++;
}
```

Another typical problem is to find the largest number in `A`. The strategy is to go through the array, keeping track of the largest number found so far. We'll store the largest number found so far in a variable called `max`. As we look through the array, whenever we find a number larger than the current value of `max`, we change the value of `max` to that larger value. After the whole array has been processed, `max` is the largest item in the array overall. The only question is, what should the original value of `max` be? One possibility is to start with `max` equal to `A[0]`, and then to look through the rest of the array, starting from `A[1]`, for larger items:

```
double max = A[0];
for (int i = 1; i < A.length; i++) {
   if (A[i] > max)
      max = A[i];
}
// at this point, max is the largest item in A
```

(There is one subtle problem here. It's possible in Java for an array to have length zero. In that case, `A[0]` doesn't exist, and the reference to `A[0]` in the first line gives an array index out of bounds error. However, zero-length arrays are normally something that you want to avoid in real problems. Anyway, what would it mean to ask for the largest item in an array that contains no items at all?)

As a final example of basic array operations, consider the problem of copying an array. To make a copy of our sample array `A`, it is **not** sufficient to say

```
double[] B = A;
```

since this does not create a new array object. All it does is declare a new array variable and make it refer to the same object to which `A` refers. (So that, for example, a change to `A[i]` will automatically change `B[i]` as well.) Remember that arrays are objects, and array variables hold pointers to objects; the assignment `B = A` just copies a pointer. To make a new array that is a copy of `A`, it is necessary to make a new array object and to copy each of the individual items from `A` into the new array:

```
double[] B = new double[A.length]; // Make a new array object,
                                   //   the same size as A.
for (int i = 0; i < A.length; i++)
   B[i] = A[i];   // Copy each item from A to B.
```

Copying values from one array to another is such a common operation that Java has a predefined subroutine to do it. The subroutine, `System.arraycopy()`, is a static method in the standard `System` class. Its declaration has the form

```
public static void arraycopy(Object sourceArray, int sourceStartIndex,
        Object destArray, int destStartIndex, int count)
```

where `sourceArray` and `destArray` can be arrays with any base type. Values are copied from `sourceArray` to `destArray`. The `count` tells how many elements to copy. Values are taken from `sourceArray` starting at position `sourceStartIndex` and are stored in `destArray` starting at position `destStartIndex`. For example, to make a copy of the array, `A`, using this subroutine, you would say:

```
double B = new double[A.length];
System.arraycopy( A, 0, B, 0, A.length );
```

7.2.2 Arrays and for-each Loops

Java 5.0 introduced a new form of the `for` loop, the "for-each loop" that was discussed in Subsection 3.4.4. The for-each loop is meant specifically for processing all the values in a data structure. When used to process an array, a for-each loop can be used to perform the same operation on each value that is stored in the array. If `anArray` is an array of type `BaseType[]`, then a for-each loop for `anArray` has the form:

```
for ( BaseType item : anArray ) {
   .
   .  // process the item
   .
}
```

In this loop, `item` is the loop control variable. It is being declared as a variable of type *BaseType*, where *BaseType* is the base type of the array. (In a for-each loop, the loop control variable **must** be declared in the loop.) When this loop is executed, each value from the array is assigned to `item` in turn and the body of the loop is executed for each value. Thus, the above loop is exactly equivalent to:

```
for ( int index = 0; index < anArray.length; index++ ) {
   BaseType item;
   item = anArray[index];  // Get one of the values from the array
     .
     .  // process the item
     .
}
```

For example, if `A` is an array of type `int[]`, then we could print all the values from `A` with the for-each loop:

```
for ( int item : A )
   System.out.println( item );
```

and we could add up all the positive integers in `A` with:

```
int sum = 0;   // This will be the sum of all the positive numbers in A
for ( int item : A ) {
   if (item > 0)
      sum = sum + item;
}
```

The for-each loop is not always appropriate. For example, there is no simple way to use it to process the items in just a part of an array. However, it does make it a little easier to process all the values in an array, since it eliminates any need to use array indices.

It's important to note that a for-each loop processes the **values** in the array, not the **elements** (where an element means the actual memory location that is part of the array). For example, consider the following incorrect attempt to fill an array of integers with 17's:

```
int[] intList = new int[10];
for ( int item : intList ) {   // INCORRECT! DOES NOT MODIFY THE ARRAY!
   item = 17;
}
```

The assignment statement `item = 17` assigns the value 17 to the loop control variable, `item`. However, this has nothing to do with the array. When the body of the loop is executed, the value from one of the elements of the array is copied into `item`. The statement `item = 17` replaces that copied value but has no effect on the array element from which it was copied; the value in the array is not changed.

7.2.3 Array Types in Subroutines

Any array type, such as `double[]`, is a full-fledged Java type, so it can be used in all the ways that any other Java type can be used. In particular, it can be used as the type of a formal parameter in a subroutine. It can even be the return type of a function. For example, it might be useful to have a function that makes a copy of an array of `double`:

```
/**
 *  Create a new array of doubles that is a copy of a given array.
 *  @param source the array that is to be copied; the value can be null
 *  @return a copy of source; if source is null, then the return value is also null
 */
public static double[]  copy( double[] source ) {
    if ( source == null )
       return null;
    double[]  cpy;  // A copy of the source array.
    cpy = new double[source.length];
    System.arraycopy( source, 0, cpy, 0, source.length );
    return cpy;
}
```

The `main()` routine of a program has a parameter of type `String[]`. You've seen this used since all the way back in Section 2.1, but I haven't really been able to explain it until now. The parameter to the `main()` routine is an array of *Strings*. When the system calls the `main()` routine, it passes an actual array of strings, which becomes the value of this parameter. Where do the strings come from? The strings in the array are the ***command-line arguments*** from the command that was used to run the program. When using a command-line interface, the user types a command to tell the system to execute a program. The user can include extra input in this command, beyond the name of the program. This extra input becomes the command-line arguments. For example, if the name of the class that contains the `main()` routine is `myProg`, then the user can type "`java myProg`" to execute the program. In this case, there are no command-line arguments. But if the user types the command

```
java myProg one two three
```

then the command-line arguments are the strings "one", "two", and "three". The system puts these strings into an array of *Strings* and passes that array as a parameter to the `main()` routine. Here, for example, is a short program that simply prints out any command line arguments entered by the user:

```
public class CLDemo {

   public static void main(String[] args) {
      System.out.println("You entered " + args.length
                                    + " command-line arguments");
      if (args.length > 0) {
         System.out.println("They were:");
         for (int i = 0; i < args.length; i++)
            System.out.println("   " + args[i]);
      }
   } // end main()

} // end class CLDemo
```

Note that the parameter, `args`, is never `null` when `main()` is called by the system, but it might be an array of length zero.

In practice, command-line arguments are often the names of files to be processed by the program. I will give some examples of this in Chapter 11, when I discuss file processing.

7.2.4 Random Access

So far, all my examples of array processing have used ***sequential access***. That is, the elements of the array were processed one after the other in the sequence in which they occur in the array. But one of the big advantages of arrays is that they allow ***random access***. That is, every element of the array is equally accessible at any given time.

As an example, let's look at a well-known problem called the birthday problem: Suppose that there are `N` people in a room. What's the chance that there are two people in the room who have the same birthday? (That is, they were born on the same day in the same month, but not necessarily in the same year.) Most people severely underestimate the probability. We will actually look at a different version of the question: Suppose you choose people at random and check their birthdays. How many people will you check before you find one who has the same birthday as someone you've already checked? Of course, the answer in a particular case depends on random factors, but we can simulate the experiment with a computer program and run the program several times to get an idea of how many people need to be checked on average.

To simulate the experiment, we need to keep track of each birthday that we find. There are 365 different possible birthdays. (We'll ignore leap years.) For each possible birthday, we need to keep track of whether or not we have already found a person who has that birthday. The answer to this question is a boolean value, true or false. To hold the data for all 365 possible birthdays, we can use an array of 365 boolean values:

```
boolean[] used;
used = new boolean[365];
```

The days of the year are numbered from 0 to 364. The value of `used[i]` is true if someone has been selected whose birthday is day number `i`. Initially, all the values in the array, `used`, are false. When we select someone whose birthday is day number `i`, we first check whether `used[i]` is true. If it is true, then this is the second person with that birthday. We are done. If `used[i]` is false, we set `used[i]` to be true to record the fact that we've encountered someone with that birthday, and we go on to the next person. Here is a subroutine that carries out the simulated experiment (of course, in the subroutine, there are no simulated people, only simulated birthdays):

```
/**
 * Simulate choosing people at random and checking the day of the year they
 * were born on.  If the birthday is the same as one that was seen previously,
 * stop, and output the number of people who were checked.
 */
private static void birthdayProblem() {

   boolean[] used;  // For recording the possible birthdays
                    //   that have been seen so far.  A value
                    //   of true in used[i] means that a person
                    //   whose birthday is the i-th day of the
                    //   year has been found.

   int count;       // The number of people who have been checked.

   used = new boolean[365];  // Initially, all entries are false.

   count = 0;

   while (true) {
          // Select a birthday at random, from 0 to 364.
```

```
            // If the birthday has already been used, quit.
            // Otherwise, record the birthday as used.
         int birthday;  // The selected birthday.
         birthday = (int)(Math.random()*365);
         count++;
         if ( used[birthday] )  // This day was found before; It's a duplicate.
            break;
         used[birthday] = true;
      }

      System.out.println("A duplicate birthday was found after "
                                              + count + " tries.");

   } // end birthdayProblem()
```

This subroutine makes essential use of the fact that every element in a newly created array of **boolean** is set to be `false`. If we wanted to reuse the same array in a second simulation, we would have to reset all the elements in it to be `false` with a `for` loop:

```
for (int i = 0; i < 365; i++)
    used[i] = false;
```

The sample program that uses this subroutine is *BirthdayProblemDemo.java*. An applet version of the program can be found in the online version of this section.

7.2.5 Arrays of Objects

One of the examples in Subsection 6.4.2 was an applet that shows multiple copies of a message in random positions, colors, and fonts. When the user clicks on the applet, the positions, colors, and fonts are changed to new random values. Like several other examples from that chapter, the applet had a flaw: It didn't have any way of storing the data that would be necessary to redraw itself. Arrays provide us with one possible solution to this problem. We can write a new version of the RandomStrings applet that uses an array to store the position, font, and color of each string. When the content pane of the applet is painted, this information is used to draw the strings, so the applet will paint itself correctly whenever it has to be redrawn. When the user clicks on the applet, the array is filled with new random values and the applet is repainted using the new data. So, the only time that the picture will change is in response to a mouse click.

In this applet, the number of copies of the message is given by a named constant, `MESSAGE_COUNT`. One way to store the position, color, and font of `MESSAGE_COUNT` strings would be to use four arrays:

```
int[] x = new int[MESSAGE_COUNT];
int[] y = new int[MESSAGE_COUNT];
Color[] color = new Color[MESSAGE_COUNT];
Font[] font = new Font[MESSAGE_COUNT];
```

These arrays would be filled with random values. In the `paintComponent()` method, the `i`-th copy of the string would be drawn at the point `(x[i],y[i])`. Its color would be given by `color[i]`. And it would be drawn in the font `font[i]`. This would be accomplished by the `paintComponent()` method

```
public void paintComponent(Graphics g) {
   super.paintComponent(); // (Fill with background color.)
   for (int i = 0; i < MESSAGE_COUNT; i++) {
      g.setColor( color[i] );
      g.setFont( font[i] );
      g.drawString( message, x[i], y[i] );
   }
}
```

This approach is said to use ***parallel arrays***. The data for a given copy of the message is spread out across several arrays. If you think of the arrays as laid out in parallel columns—array `x` in the first column, array `y` in the second, array `color` in the third, and array `font` in the fourth—then the data for the `i`-th string can be found along the `i`-th row. There is nothing wrong with using parallel arrays in this simple example, but it does go against the object-oriented philosophy of keeping related data in one object. If we follow this rule, then we don't have to **imagine** the relationship among the data, because all the data for one copy of the message is physically in one place. So, when I wrote the applet, I made a simple class to represent all the data that is needed for one copy of the message:

```
/**
 * An object of this type holds the position, color, and font
 * of one copy of the string.
 */
private static class StringData {
   int x, y;     // The coordinates of the left end of baseline of string.
   Color color;  // The color in which the string is drawn.
   Font font;    // The font that is used to draw the string.
}
```

(This class is actually defined as a static nested class in the main applet class.) To store the data for multiple copies of the message, I use an array of type StringData[]. The array is declared as an instance variable, with the name `stringData`:

```
StringData[] stringData;
```

Of course, the value of `stringData` is `null` until an actual array is created and assigned to it. This is done in the `init()` method of the applet with the statement

```
stringData = new StringData[MESSAGE_COUNT];
```

The base type of this array is *StringData*, which is a class. We say that `stringData` is an ***array of objects***. This means that the elements of the array are variables of type *StringData*. Like any object variable, each element of the array can either be `null` or can hold a reference to an object. (Note that the term "array of objects" is a little misleading, since the objects are not in the array; the array can only contain references to objects.) When the `stringData` array is first created, the value of each element in the array is `null`.

The data needed by the RandomStrings program will be stored in objects of type *StringData*, but no such objects exist yet. All we have so far is an array of variables that are capable of referring to such objects. I decided to create the *StringData* objects in the applet's `init` method. (It could be done in other places—just so long as we avoid trying to use an object that doesn't exist. This is important: Remember that a newly created array whose base type is an object type is always filled with `null` elements. There are **no** objects in the array until you put them there.) The objects are created with the `for` loop

```
for (int i = 0; i < MESSAGE_COUNT; i++)
    stringData[i] = new StringData();
```

For the RandomStrings applet, the idea is to store data for the `i`-th copy of the message in the variables `stringData[i].x`, `stringData[i].y`, `stringData[i].color`, and `stringData[i].font`. Make sure that you understand the notation here: `stringData[i]` refers to an object. That object contains instance variables. The notation `stringData[i].x` tells the computer: "Find your way to the object that is referred to by `stringData[i]`. Then go to the instance variable named `x` in that object." Variable names can get even more complicated than this, so it is important to learn how to read them. Using the array, `stringData`, the `paintComponent()` method for the applet could be written

```
public void paintComponent(Graphics g) {
   super.paintComponent(g); // (Fill with background color.)
   for (int i = 0; i < MESSAGE_COUNT; i++) {
      g.setColor( stringData[i].color );
      g.setFont( stringData[i].font );
      g.drawString( message, stringData[i].x, stringData[i].y );
   }
}
```

However, since the `for` loop is processing every value in the array, an alternative would be to use a for-each loop:

```
public void paintComponent(Graphics g) {
   super.paintComponent(g);
   for ( StringData data : stringData) {
          // Draw a copy of the message in the position, color,
          // and font stored in data.
      g.setColor( data.color );
      g.setFont( data.font );
      g.drawString( message, data.x, data.y );
   }
}
```

In this loop, the loop control variable, `data`, holds a copy of one of the values from the array. That value is a reference to an object of type *StringData*, which has instance variables named `color`, `font`, `x`, and `y`. Once again, the use of a for-each loop has eliminated the need to work with array indices.

There is still the matter of filling the array, `data`, with random values. If you are interested, you can look at the source code for the applet, *RandomStringsWithArray.java*.

* * *

The RandomStrings applet uses one other array of objects. The font for a given copy of the message is chosen at random from a set of five possible fonts. In the original version of the applet, there were five variables of type *Font* to represent the fonts. The variables were named `font1`, `font2`, `font3`, `font4`, and `font5`. To select one of these fonts at random, a `switch` statement could be used:

```
Font randomFont;  // One of the 5 fonts, chosen at random.
int rand;         // A random integer in the range 0 to 4.

rand = (int)(Math.random() * 5);
switch (rand) {
   case 0:
```

```
            randomFont = font1;
            break;
         case 1:
            randomFont = font2;
            break;
         case 2:
            randomFont = font3;
            break;
         case 3:
            randomFont = font4;
            break;
         case 4:
            randomFont = font5;
            break;
      }
```

In the new version of the applet, the five fonts are stored in an array, which is named `fonts`. This array is declared as an instance variable of type Font[]

```
Font[] fonts;
```

The array is created in the `init()` method of the applet, and each element of the array is set to refer to a new *Font* object:

```
fonts = new Font[5];  // Create the array to hold the five fonts.

fonts[0] = new Font("Serif", Font.BOLD, 14);
fonts[1] = new Font("SansSerif", Font.BOLD + Font.ITALIC, 24);
fonts[2] = new Font("Monospaced", Font.PLAIN, 20);
fonts[3] = new Font("Dialog", Font.PLAIN, 30);
fonts[4] = new Font("Serif", Font.ITALIC, 36);
```

This makes it much easier to select one of the fonts at random. It can be done with the statements

```
Font randomFont;  // One of the 5 fonts, chosen at random.
int fontIndex;    // A random number in the range 0 to 4.
fontIndex = (int)(Math.random() * 5);
randomFont = fonts[ fontIndex ];
```

The `switch` statement has been replaced by a single line of code. In fact, the preceding four lines could be replaced by the single line:

```
Font randomFont = fonts[ (int)(Math.random() * 5) ];
```

This is a very typical application of arrays. Note that this example uses the random access property of arrays: We can pick an array index at random and go directly to the array element at that index.

Here is another example of the same sort of thing. Months are often stored as numbers 1, 2, 3, . . . , 12. Sometimes, however, these numbers have to be translated into the names January, February, . . . , December. The translation can be done with an array. The array can be declared and initialized as

```
static String[] monthName = { "January", "February", "March",
                              "April",   "May",      "June",
                              "July",    "August",   "September",
                              "October", "November", "December" };
```

If `mnth` is a variable that holds one of the integers 1 through 12, then `monthName[mnth-1]` is the name of the corresponding month. We need the "`-1`" because months are numbered starting from 1, while array elements are numbered starting from 0. Simple array indexing does the translation for us!

7.2.6 Variable Arity Methods

Arrays are used in the implementation of a feature that was introduced in Java 5.0. Before version 5.0, every method in Java had a fixed arity. (The ***arity*** of a subroutine is defined as the number of parameters in a call to the method.) In a fixed arity method, the number of parameters must be the same in every call to the method. Java 5.0 introduced ***variable arity methods***. In a variable arity method, different calls to the method can have different numbers of parameters. For example, the formatted output method `System.out.printf`, which was introduced in Subsection 2.4.4, is a variable arity method. The first parameter of `System.out.printf` must be a *String*, but it can have any number of additional parameters, of any types.

Calling a variable arity method is no different from calling any other sort of method, but writing one requires some new syntax. As an example, consider a method that can compute the average of any number of values of type **double**. The definition of such a method could begin with:

```
public static double average( double...  numbers ) {
```

Here, the `...` after the type name, `double`, indicates that any number of values of type **double** can be provided when the subroutine is called, so that for example `average(1,4,9,16)`, `average(3.14,2.17)`, `average(0.375)`, and even `average()` are all legal calls to this method. Note that actual parameters of type **int** can be passed to `average`. The integers will, as usual, be automatically converted to real numbers.

When the method is called, the values of all the actual parameters that correspond to the variable arity parameter are placed into an array, and it is this array that is actually passed to the method. That is, in the body of a method, a variable arity parameter of type *T* actually looks like an ordinary parameter of type `T[]`. The length of the array tells you how many actual parameters were provided in the method call. In the `average` example, the body of the method would see an array named `numbers` of type `double[]`. The number of actual parameters in the method call would be `numbers.length`, and the values of the actual parameters would be `numbers[0]`, `numbers[1]`, and so on. A complete definition of the method would be:

```
public static double average( double... numbers ) {
   double sum;      // The sum of all the actual parameters.
   double average;  // The average of all the actual parameters.
   sum = 0;
   for (int i = 0; i < numbers.length; i++) {
      sum = sum + numbers[i];  // Add one of the actual parameters to the sum.
   }
   average = sum / numbers.length;
   return average;
}
```

Note that the "`...`" can be applied only to the **last** formal parameter in a method definition. Note also that it is possible to pass an actual array to the method, instead of a list of individual values. For example, if `salesData` is a variable of type `double[]`, then it would be legal to call `average(salesData)`, and this would compute the average of all the numbers in the array.

As another example, consider a method that can draw a polygon through any number of points. The points are given as values of type *Point*, where an object of type *Point* has two instance variables, `x` and `y`, of type **int**. In this case, the method has one ordinary parameter—the graphics context that will be used to draw the polygon—in addition to the variable arity parameter:

```
public static void drawPolygon(Graphics g, Point... points) {
    if (points.length > 1) {  // (Need at least 2 points to draw anything.)
       for (int i = 0; i < points.length - 1; i++) {
           // Draw a line from i-th point to (i+1)-th point
           g.drawLine( points[i].x, points[i].y, points[i+1].x, points[i+1].y );
       }
       // Now, draw a line back to the starting point.
       g.drawLine( points[points.length-1].x, points[points.length-1].y,
                       points[0].x, points[0].y );
    }
}
```

Because of automatic type conversion, a variable arity parameter of type "`Object...`" can take actual parameters of any type whatsoever. Even primitive type values are allowed, because of autoboxing. (A primitive type value belonging to a type such as **int** is converted to an object belonging to a "wrapper" class such as *Integer*. See Subsection 5.3.2.) For example, the method definition for `System.out.printf` could begin:

```
public void printf(String format, Object... values) {
```

This allows the `printf` method to output values of any type. Similarly, we could write a method that strings together the string representations of all its parameters into one long string:

```
public static String concat( Object... values ) {
   StringBuffer buffer;  // Use a StringBuffer for more efficient concatenation.
   buffer = new StringBuffer();  // Start with an empty buffer.
   for ( Object obj : values ) { // A "for each" loop for processing the values.
       buffer.append(obj); // Add string representation of obj to the buffer.
   }
   return buffer.toString(); // return the contents of the buffer
}
```

7.3 Dynamic Arrays and ArrayLists

The size of an array is fixed when it is created. In many cases, however, the number of data items that are actually stored in the array varies with time. Consider the following examples: An array that stores the lines of text in a word-processing program. An array that holds the list of computers that are currently downloading a page from a Web site. An array that contains the shapes that have been added to the screen by the user of a drawing program. Clearly, we need some way to deal with cases where the number of data items in an array is not fixed.

7.3.1 Partially Full Arrays

Consider an application where the number of items that we want to store in an array changes as the program runs. Since the size of the array can't actually be changed, a separate counter variable must be used to keep track of how many spaces in the array are in use. (Of course,

every space in the array has to contain something; the question is, how many spaces contain useful or valid items?)

Consider, for example, a program that reads positive integers entered by the user and stores them for later processing. The program stops reading when the user inputs a number that is less than or equal to zero. The input numbers can be kept in an array, `numbers`, of type `int[]`. Let's say that no more than 100 numbers will be input. Then the size of the array can be fixed at 100. But the program must keep track of how many numbers have actually been read and stored in the array. For this, it can use an integer variable, `numCount`. Each time a number is stored in the array, `numCount` must be incremented by one. As a rather silly example, let's write a program that will read the numbers input by the user and then print them in the reverse of the order in which they were entered. (This is, at least, a processing task that requires that the numbers be saved in an array. Remember that many types of processing, such as finding the sum or average or maximum of the numbers, can be done without saving the individual numbers.)

```
public class ReverseInputNumbers {

   public static void main(String[] args) {

     int[] numbers;  // An array for storing the input values.
     int numCount;   // The number of numbers saved in the array.
     int num;        // One of the numbers input by the user.

     numbers = new int[100];   // Space for 100 ints.
     numCount = 0;             // No numbers have been saved yet.

     TextIO.putln("Enter up to 100 positive integers; enter 0 to end.");

     while (true) {   // Get the numbers and put them in the array.
        TextIO.put("? ");
        num = TextIO.getlnInt();
        if (num <= 0)
           break;
        numbers[numCount] = num;
        numCount++;
     }

     TextIO.putln("\nYour numbers in reverse order are:\n");

     for (int i = numCount - 1; i >= 0; i--) {
         TextIO.putln( numbers[i] );
     }

   } // end main();

}  // end class ReverseInputNumbers
```

It is especially important to note that the variable `numCount` plays a dual role. It is the number of items that have been entered into the array. But it is also the index of the next available spot in the array. For example, if 4 numbers have been stored in the array, they occupy locations number 0, 1, 2, and 3. The next available spot is location 4. When the time comes to print out the numbers in the array, the last occupied spot in the array is location `numCount - 1`, so the `for` loop prints out values starting from location `numCount - 1` and going down to 0.

Let's look at another, more realistic example. Suppose that you write a game program, and that players can join the game and leave the game as it progresses. As a good object-oriented

programmer, you probably have a class named *Player* to represent the individual players in the game. A list of all players who are currently in the game could be stored in an array, `playerList`, of type `Player[]`. Since the number of players can change, you will also need a variable, `playerCt`, to record the number of players currently in the game. Assuming that there will never be more than 10 players in the game, you could declare the variables as:

```
Player[] playerList = new Player[10];  // Up to 10 players.
int      playerCt = 0;  // At the start, there are no players.
```

After some players have joined the game, `playerCt` will be greater than 0, and the player objects representing the players will be stored in the array elements `playerList[0]`, `playerList[1]`, . . . , `playerList[playerCt-1]`. Note that the array element `playerList[playerCt]` is **not** in use. The procedure for adding a new player, `newPlayer`, to the game is simple:

```
playerList[playerCt] = newPlayer; // Put new player in next
                                  //     available spot.
playerCt++;  // And increment playerCt to count the new player.
```

Deleting a player from the game is a little harder, since you don't want to leave a "hole" in the array. Suppose you want to delete the player at index `k` in `playerList`. If you are not worried about keeping the players in any particular order, then one way to do this is to move the player from the last occupied position in the array into position `k` and then to decrement the value of `playerCt`:

```
playerList[k] = playerList[playerCt - 1];
playerCt--;
```

The player previously in position `k` is no longer in the array. The player previously in position `playerCt - 1` is now in the array twice. But it's only in the occupied or valid part of the array once, since `playerCt` has decreased by one. Remember that every element of the array has to hold some value, but only the values in positions 0 through `playerCt - 1` will be looked at or processed in any way. (By the way, you should think about what happens if the player that is being deleted is in the last position in the list. The code does still work in this case. What exactly happens?)

Suppose that when deleting the player in position `k`, you'd like to keep the remaining players in the same order. (Maybe because they take turns in the order in which they are stored in the array.) To do this, all the players in positions `k+1` and above must move down one position in the array. Player `k+1` replaces player `k`, who is out of the game. Player `k+2` fills the spot left open when player `k+1` is moved. And so on. The code for this is

```
for (int i = k+1; i < playerCt; i++) {
    playerList[i-1] = playerList[i];
}
playerCt--;
```

* * *

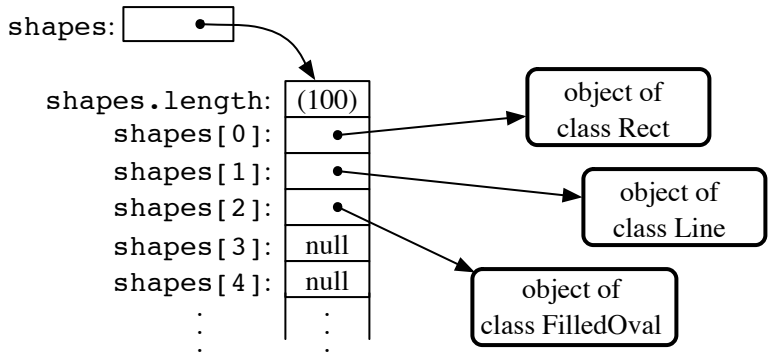
It's worth emphasizing that the *Player* example deals with an array whose base type is a class. An item in the array is either `null` or is a reference to an object belonging to the class, *Player*. The *Player* objects themselves are not really stored in the array, only references to them. Note that because of the rules for assignment in Java, the objects can actually belong to subclasses of *Player*. Thus there could be different classes of players such as computer players,

regular human players, players who are wizards, . . . , all represented by different subclasses of *Player*.

As another example, suppose that a class *Shape* represents the general idea of a shape drawn on a screen, and that it has subclasses to represent specific types of shapes such as lines, rectangles, rounded rectangles, ovals, filled-in ovals, and so forth. (*Shape* itself would be an abstract class, as discussed in Subsection 5.5.5.) Then an array of type Shape[] can hold references to objects belonging to the subclasses of *Shape*. For example, the situation created by the statements

```
Shape[] shapes = new Shape[100]; // Array to hold up to 100 shapes.
shapes[0] = new Rect();          // Put some objects in the array.
shapes[1] = new Line();
shapes[2] = new FilledOval();
int shapeCt = 3;  // Keep track of number of objects in array.
```

could be illustrated as:



Such an array would be useful in a drawing program. The array could be used to hold a list of shapes to be displayed. If the `Shape` class includes a method, "`void redraw(Graphics g)`", for drawing the shape in a graphics context `g`, then all the shapes in the array could be redrawn with a simple for loop:

```
for (int i = 0; i < shapeCt; i++)
   shapes[i].redraw(g);
```

The statement "`shapes[i].redraw(g);`" calls the `redraw()` method belonging to the particular shape at index `i` in the array. Each object knows how to redraw itself, so that repeated executions of the statement can produce a variety of different shapes on the screen. This is nice example both of polymorphism and of array processing.

7.3.2 Dynamic Arrays

In each of the above examples, an arbitrary limit was set on the number of items—100 **ints**, 10 *Players*, 100 *Shapes*. Since the size of an array is fixed, a given array can only hold a certain maximum number of items. In many cases, such an arbitrary limit is undesirable. Why should a program work for 100 data values, but not for 101? The obvious alternative of making an array that's so big that it will work in any practical case is not usually a good solution to the problem. It means that in most cases, a lot of computer memory will be wasted on unused space in the array. That memory might be better used for something else. And what if someone is

using a computer that could handle as many data values as the user actually wants to process, but doesn't have enough memory to accommodate all the extra space that you've allocated for your huge array?

Clearly, it would be nice if we could increase the size of an array at will. This is not possible, but what **is** possible is almost as good. Remember that an array variable does not actually hold an array. It just holds a reference to an array object. We can't make the array bigger, but we can make a new, bigger array object and change the value of the array variable so that it refers to the bigger array. Of course, we also have to copy the contents of the old array into the new array. The array variable then refers to an array object that contains all the data of the old array, with room for additional data. The old array will be garbage collected, since it is no longer in use.

Let's look back at the game example, in which `playerList` is an array of type Player[] and `playerCt` is the number of spaces that have been used in the array. Suppose that we don't want to put a pre-set limit on the number of players. If a new player joins the game and the current array is full, we just make a new, bigger one. The same variable, `playerList`, will refer to the new array. Note that after this is done, `playerList[0]` will refer to a different memory location, but the value stored in `playerList[0]` will still be the same as it was before. Here is some code that will do this:

```
// Add a new player, even if the current array is full.

if (playerCt == playerList.length) {
        // Array is full.  Make a new, bigger array,
        // copy the contents of the old array into it,
        // and set playerList to refer to the new array.
   int newSize = 2 * playerList.length;  // Size of new array.
   Player[] temp = new Player[newSize];  // The new array.
   System.arraycopy(playerList, 0, temp, 0, playerList.length);
   playerList = temp;  // Set playerList to refer to new array.
}

// At this point, we KNOW there is room in the array.

playerList[playerCt] = newPlayer; // Add the new player...
playerCt++;                       //    ...and count it.
```

If we are going to be doing things like this regularly, it would be nice to define a reusable class to handle the details. An array-like object that changes size to accommodate the amount of data that it actually contains is called a ***dynamic array***. A dynamic array supports the same operations as an array: putting a value at a given position and getting the value that is stored at a given position. But there is no upper limit on the positions that can be used (except those imposed by the size of the computer's memory). In a dynamic array class, the `put` and `get` operations must be implemented as instance methods. Here, for example, is a class that implements a dynamic array of **ints**:

```
/**
 *  An object of type DynamicArrayOfInt acts like an array of int
 *  of unlimited size.  The notation A.get(i) must be used instead
 *  of A[i], and A.set(i,v) must be used instead of A[i] = v.
 */
public class DynamicArrayOfInt {

   private int[] data;  // An array to hold the data.
```

```
   /**
    * Constructor creates an array with an initial size of 1,
    * but the array size will be increased whenever a reference
    * is made to an array position that does not yet exist.
    */
   public DynamicArrayOfInt() {
      data = new int[1];
   }

   /**
    *  Get the value from the specified position in the array.
    *  Since all array elements are initialized to zero, when the
    *  specified position lies outside the actual physical size
    *  of the data array, a value of 0 is returned.  Note that
    *  a negative value of position will still produce an
    *  ArrayIndexOutOfBoundsException.
    */
   public int get(int position) {
      if (position >= data.length)
         return 0;
      else
         return data[position];
   }

   /**
    *  Store the value in the specified position in the array.
    *  The data array will increase in size to include this
    *  position, if necessary.
    */
   public void put(int position, int value) {
      if (position >= data.length) {
             // The specified position is outside the actual size of
             // the data array.  Double the size, or if that still does
             // not include the specified position, set the new size
             // to 2*position.
         int newSize = 2 * data.length;
         if (position >= newSize)
            newSize = 2 * position;
         int[] newData = new int[newSize];
         System.arraycopy(data, 0, newData, 0, data.length);
         data = newData;
             // The following line is for demonstration purposes only !!
         System.out.println("Size of dynamic array increased to " + newSize);
      }
      data[position] = value;
   }

} // end class DynamicArrayOfInt
```

The data in a *DynamicArrayOfInt* object is actually stored in a regular array, but that array is discarded and replaced by a bigger array whenever necessary. If `numbers` is a variable of type *DynamicArrayOfInt*, then the command `numbers.put(pos,val)` stores the value `val` at position number `pos` in the dynamic array. The function `numbers.get(pos)` returns the value stored at position number `pos`.

The first example in this section used an array to store positive integers input by the user. We can rewrite that example to use a *DynamicArrayOfInt*. A reference to `numbers[i]` is replaced by `numbers.get(i)`. The statement "`numbers[numCount] = num;`" is replaced by "`numbers.put(numCount,num);`". Here's the program:

```
public class ReverseWithDynamicArray {

   public static void main(String[] args) {

      DynamicArrayOfInt numbers;  // To hold the input numbers.
      int numCount;  // The number of numbers stored in the array.
      int num;    // One of the numbers input by the user.

      numbers = new DynamicArrayOfInt();
      numCount = 0;

      TextIO.putln("Enter some positive integers; Enter 0 to end");
      while (true) {  // Get numbers and put them in the dynamic array.
         TextIO.put("? ");
         num = TextIO.getlnInt();
         if (num <= 0)
            break;
         numbers.put(numCount, num);  // Store num in the dynamic array.
         numCount++;
      }

      TextIO.putln("\nYour numbers in reverse order are:\n");

      for (int i = numCount - 1; i >= 0; i--) {
          TextIO.putln( numbers.get(i) );  // Print the i-th number.
      }

   } // end main();

}  // end class ReverseWithDynamicArray
```

You can find an applet that simulates the program in the on-line version of this section.

7.3.3 ArrrayLists

The *DynamicArrayOfInt* class could be used in any situation where an array of **int** with no preset limit on the size is needed. However, if we want to store *Shapes* instead of **ints**, we would have to define a new class to do it. That class, probably named "*DynamicArrayOfShape*", would look exactly the same as the *DynamicArrayOfInt* class except that everywhere the type "**int**" appears, it would be replaced by the type "*Shape*". Similarly, we could define a *DynamicArrayOfDouble* class, a *DynamicArrayOfPlayer* class, and so on. But there is something a little silly about this, since all these classes are close to being identical. It would be nice to be able to write some kind of source code, once and for all, that could be used to generate any of these classes on demand, given the type of value that we want to store. This would be an example of ***generic programming***. Some programming languages, including C++, have had support for generic programming for some time. With version 5.0, Java introduced true generic programming, but even before that it had something that was very similar: One can come close to generic programming in Java by working with data structures that contain elements of type *Object*. We will first consider the almost-generic programming that has been available in Java from

the beginning, and then we will look at the change that was introduced in Java 5.0. A full discussion of generic programming will be given in Chapter 10.

In Java, every class is a subclass of the class named *Object*. This means that every object can be assigned to a variable of type *Object*. Any object can be put into an array of type Object[]. If we defined a *DynamicArrayOfObject* class, then we could store objects of any type. This is not true generic programming, and it doesn't apply to the primitive types such as **int** and **double**. But it does come close. In fact, there is no need for us to define a *DynamicArrayOfObject* class. Java already has a standard class named *ArrayList* that serves much the same purpose. The *ArrayList* class is in the package `java.util`, so if you want to use it in a program, you should put the directive "`import java.util.ArrayList;`" at the beginning of your source code file.

The *ArrayList* class differs from my *DynamicArrayOfInt* class in that an *ArrayList* object always has a definite size, and it is illegal to refer to a position in the *ArrayList* that lies outside its size. In this, an *ArrayList* is more like a regular array. However, the size of an *ArrayList* can be increased at will. The *ArrayList* class defines many instance methods. I'll describe some of the most useful. Suppose that `list` is a variable of type *ArrayList*. Then we have:

- `list.size()` — This function returns the current size of the *ArrayList*. The only valid positions in the list are numbers in the range `0` to `list.size()-1`. Note that the size can be zero. A call to the default constructor `new ArrayList()` creates an *ArrayList* of size zero.
- `list.add(obj)` — Adds an object onto the end of the list, increasing the size by 1. The parameter, `obj`, can refer to an object of any type, or it can be `null`.
- `list.get(N)` — This function returns the value stored at position `N` in the *ArrayList*. `N` must be an integer in the range `0` to `list.size()-1`. If `N` is outside this range, an error of type *IndexOutOfBoundsException* occurs. Calling this function is similar to referring to `A[N]` for an array, `A`, except that you can't use `list.get(N)` on the left side of an assignment statement.
- `list.set(N, obj)` — Assigns the object, `obj`, to position `N` in the *ArrayList*, replacing the item previously stored at position `N`. The integer `N` must be in the range from `0` to `list.size()-1`. A call to this function is equivalent to the command `A[N] = obj` for an array `A`.
- `list.remove(obj)` — If the specified object occurs somewhere in the *ArrayList*, it is removed from the list. Any items in the list that come after the removed item are moved down one position. The size of the *ArrayList* decreases by 1. If `obj` occurs more than once in the list, only the first copy is removed.
- `list.remove(N)` — For an integer, `N`, this removes the `N-th` item in the *ArrayList*. `N` must be in the range `0` to `list.size()-1`. Any items in the list that come after the removed item are moved down one position. The size of the *ArrayList* decreases by 1.
- `list.indexOf(obj)` — A function that searches for the object, `obj`, in the *ArrayList*. If the object is found in the list, then the position number where it is found is returned. If the object is not found, then `-1` is returned.

For example, suppose again that players in a game are represented by objects of type *Player*. The players currently in the game could be stored in an *ArrayList* named `players`. This variable would be declared as

```
ArrayList players;
```

and initialized to refer to a new, empty *ArrayList* object with

```
players = new ArrayList();
```

If `newPlayer` is a variable that refers to a *Player* object, the new player would be added to the *ArrayList* and to the game by saying

```
players.add(newPlayer);
```

and if player number `i` leaves the game, it is only necessary to say

```
players.remove(i);
```

Or, if `player` is a variable that refers to the *Player* that is to be removed, you could say

```
players.remove(player);
```

All this works very nicely. The only slight difficulty arises when you use the function `players.get(i)` to get the value stored at position `i` in the *ArrayList*. The return type of this function is *Object*. In this case the object that is returned by the function is actually of type *Player*. In order to do anything useful with the returned value, it's usually necessary to type-cast it to type *Player*:

```
Player plr = (Player)players.get(i);
```

For example, if the *Player* class includes an instance method `makeMove()` that is called to allow a player to make a move in the game, then the code for letting every player make a move is

```
for (int i = 0;  i < players.size();  i++) {
    Player plr = (Player)players.get(i);
    plr.makeMove();
}
```

The two lines inside the `for` loop can be combined into a single line:

```
((Player)players.get(i)).makeMove();
```

This gets an item from the list, type-casts it, and then calls the `makeMove()` method on the resulting *Player*. The parentheses around "`(Player)players.get(i)`" are required because of Java's precedence rules. The parentheses force the type-cast to be performed before the `makeMove()` method is called.

For-each loops work for *ArrayLists* just as they do for arrays. But note that since the items in an *ArrayList* are only known to be *Objects*, the type of the loop control variable must be *Object*. For example, the `for` loop used above to let each *Player* make a move could be written as the for-each loop

```
for ( Object plrObj : players ) {
   Player plr = (Player)plrObj;
   plr.makeMove();
}
```

In the body of the loop, the value of the loop control variable, `plrObj`, is one of the objects from the list, `players`. This object must be type-cast to type *Player* before it can be used.

* * *

In Subsection 5.5.5, I discussed a program, *ShapeDraw*, that uses *ArrayLists*. Here is another version of the same idea, simplified to make it easier to see how *ArrayList* is being used. The program supports the following operations: Click the large white drawing area to add a colored rectangle. (The color of the rectangle is given by a "rainbow palette" along the bottom of the applet; click the palette to select a new color.) Drag rectangles using the right mouse button.

Hold down the Alt key and click on a rectangle to delete it (or click it with the middle mouse button). Shift-click a rectangle to move it out in front of all the other rectangles. You can try an applet version of the program in the on-line version of this section.

Source code for the main panel for this program can be found in *SimpleDrawRects.java*. You should be able to follow the source code in its entirety. (You can also take a look at the file *RainbowPalette.java*, which defines the color palette shown at the bottom of the applet, if you like.) Here, I just want to look at the parts of the program that use an *ArrayList*.

The applet uses a variable named `rects`, of type *ArrayList*, to hold information about the rectangles that have been added to the drawing area. The objects that are stored in the list belong to a static nested class, *ColoredRect*, that is defined as

```
/**
 * An object of type ColoredRect holds the data for one colored rectangle.
 */
private static class ColoredRect {
   int x,y;           // Upper left corner of the rectangle.
   int width,height;  // Size of the rectangle.
   Color color;       // Color of the rectangle.
}
```

If `g` is a variable of type `Graphics`, then the following code draws all the rectangles that are stored in the list `rects` (with a black outline around each rectangle):

```
for (int i = 0;  i < rects.size();  i++) {
   ColoredRect rect = (ColoredRect)rects.get(i);
   g.setColor( rect.color );
   g.fillRect( rect.x, rect.y, rect.width, rect.height);
   g.setColor( Color.BLACK );
   g.drawRect( rect.x, rect.y, rect.width - 1, rect.height - 1);
}
```

The `i`-th rectangle in the list is obtained by calling `rects.get(i)`. Since this method returns a value of type *Object*, the return value must be typecast to its actual type, *ColoredRect*, to get access to the data that it contains.

To implement the mouse operations, it must be possible to find the rectangle, if any, that contains the point where the user clicked the mouse. To do this, I wrote the function

```
/**
 * Find the topmost rect that contains the point (x,y). Return null
 * if no rect contains that point.  The rects in the ArrayList are
 * considered in reverse order so that if one lies on top of another,
 * the one on top is seen first and is returned.
 */
ColoredRect findRect(int x, int y) {

   for (int i = rects.size() - 1;  i >= 0;  i--) {
      ColoredRect rect = (ColoredRect)rects.get(i);
      if ( x >= rect.x && x < rect.x + rect.width
               && y >= rect.y && y < rect.y + rect.height )
      return rect;  // (x,y) is inside this rect.
   }

   return null;  // No rect containing (x,y) was found.

}
```

The code for removing a `ColoredRect`, `rect`, from the drawing area is simply `rects.remove(rect)` (followed by a `repaint()`). Bringing a given rectangle out in front of all the other rectangles is just a little harder. Since the rectangles are drawn in the order in which they occur in the *ArrayList*, the rectangle that is in the last position in the list is in front of all the other rectangles on the screen. So we need to move the selected rectangle to the last position in the list. This can most easily be done in a slightly tricky way using built-in *ArrayList* operations: The rectangle is simply removed from its current position in the list and then added back at the end of the list:

```
void bringToFront(ColoredRect rect) {
   if (rect != null) {
      rects.remove(rect); // Remove rect from the list.
      rects.add(rect);    // Add it back; it will be placed in the last position.
      repaint();
   }
}
```

This should be enough to give you the basic idea. You can look in the source code for more details.

7.3.4 Parameterized Types

The main difference between true generic programming and the *ArrayList* examples in the previous subsection is the use of the type *Object* as the basic type for objects that are stored in a list. This has at least two unfortunate consequences: First, it makes it necessary to use type-casting in almost every case when an element is retrieved from that list. Second, since any type of object can legally be added to the list, there is no way for the compiler to detect an attempt to add the wrong type of object to the list; the error will be detected only at run time when the object is retrieved from the list and the attempt to type-cast the object fails. Compare this to arrays. An array of type BaseType[] can **only** hold objects of type *BaseType*. An attempt to store an object of the wrong type in the array will be detected by the compiler, and there is no need to type-cast items that are retrieved from the array back to type *BaseType*.

To address this problem, Java 5.0 introduced ***parameterized types***. *ArrayList* is an example: Instead of using the plain "ArrayList" type, it is possible to use ArrayList<BaseType>, where *BaseType* is any object type, that is, the name of a class or of an interface. (*BaseType* **cannot** be one of the primitive types.) ArrayList<BaseType> can be used to create lists that can hold only objects of type *BaseType*. For example,

```
ArrayList<ColoredRect> rects;
```

declares a variable named `rects` of type ArrayList<ColoredRect>, and

```
rects  =  new ArrayList<ColoredRect>();
```

sets `rects` to refer to a newly created list that can only hold objects belonging to the class *ColoredRect* (or to a subclass). The funny-looking name "ArrayList<ColoredRect>" is being used here in exactly the same way as an ordinary class name—don't let the "<ColoredRect>" confuse you; it's just part of the name of the type, just as it would be in the array type ColoredRect[]. When a statement such as `rects.add(x);` occurs in the program, the compiler can check whether `x` is in fact of type *ColoredRect*. If not, the compiler will report a syntax error. When an object is retrieved from the list, the compiler knows that the object must be of type *ColoredRect*, so no type-cast is necessary. You can say simply:

```
ColoredRect rect = rects.get(i)
```

You can even refer directly to an instance variable in the object, such as `rects.get(i).color`. This makes using ArrayList<ColoredRect> very similar to using ColoredRect[], with the added advantage that the list can grow to any size. Note that if a for-each loop is used to process the items in `rects`, the type of the loop control variable can be *ColoredRect*, and no type-cast is necessary. For example, when using ArrayList<ColoredRect> as the type for the list `rects`, the code for drawing all the rectangles in the list could be rewritten as:

```
for ( ColoredRect rect : rects ) {
   g.setColor( rect.color );
   g.fillRect( rect.x, rect.y, rect.width, rect.height );
   g.setColor( Color.BLACK );
   g.drawRect( rect.x, rect.y, rect.width - 1, rect.height - 1 );
}
```

You can use ArrayList<ColoredRect> anyplace where you could use a normal type: to declare variables, as the type of a formal parameter in a subroutine, or as the return type of a subroutine. You can even create a subclass of ArrayList<ColoredRect>! (Nevertheless, technically speaking, ArrayList<ColoredRect> is not considered to be a separate class from *ArrayList*. An object of type ArrayList<ColoredRect> actually belongs to the class *ArrayList*, but the compiler restricts the type of objects that can be added to the list.)

The only drawback to using parameterized types is that the base type cannot be a primitive type. For example, there is no such thing as "`ArrayList<int>`". However, this is not such a big drawback as it might seem at first, because of the "wrapper types" and "autoboxing" that were introduced in Subsection 5.3.2. A wrapper type such as *Double* or *Integer* can be used as a base type for a parameterized type. An object of type ArrayList<Double> can hold objects of type *Double*. Since each object of type *Double* holds a value of type **double**, it's almost like having a list of **doubles**. If `numlist` is declared to be of type ArrayList<Double> and if `x` is of type **double**, then the value of `x` can be added to the list by saying:

```
numlist.add( new Double(x) );
```

Furthermore, because of autoboxing, the compiler will automatically do **double**-to-*Double* and *Double*-to-**double** type conversions when necessary. This means that the compiler will treat "`numlist.add(x)`" as being equivalent to "`numlist.add( new Double(x) )`". So, behind the scenes, "`numlist.add(x)`" is actually adding an object to the list, but it looks a lot as if you are working with a list of **doubles**.

* * *

The sample program *SimplePaint2.java* demonstrates the use of parameterized types. In this program, the user can sketch curves in a drawing area by clicking and dragging with the mouse. The curves can be of any color, and the user can select the drawing color using a menu. The background color of the drawing area can also be selected using a menu. And there is a "Control" menu that contains several commands: An "Undo" command, which removes the most recently drawn curve from the screen, a "Clear" command that removes all the curves, and a "Use Symmetry" checkbox that turns a symmetry feature on and off. Curves that are drawn by the user when the symmetry option is on are reflected horizontally and vertically to produce a symmetric pattern. You can try an applet version of the program in the on-line version of this section.

Unlike the original SimplePaint program in Subsection 6.4.4, this new version uses a data structure to store information about the picture that has been drawn by the user. This data

is used in the `paintComponent()` method to redraw the picture whenever necessary. Thus, the picture doesn't disappear when, for example, the picture is covered and then uncovered. The data structure is implemented using *ArrayLists*.

The main data for a curve consists of a list of the points on the curve. This data can be stored in an object of type ArrayList<Point>, where `java.awt.Point` is one of Java's standard classes. (A *Point* object contains two public integer variables `x` and `y` that represent the coordinates of a point.) However, to redraw the curve, we also need to know its color, and we need to know whether the symmetry option should be applied to the curve. All the data that is needed to redraw the curve can be grouped into an object of type *CurveData* that is defined as

```
private static class CurveData {
   Color color;  // The color of the curve.
   boolean symmetric;  // Are horizontal and vertical reflections also drawn?
   ArrayList<Point> points;  // The points on the curve.
}
```

However, a picture can contain many curves, not just one, so to store all the data necessary to redraw the entire picture, we need a **list** of objects of type *CurveData*. For this list, we can use a variable `curves` declared as

```
ArrayList<CurveData> curves = new ArrayList<CurveData>();
```

Here we have a list of objects, where each object contains a list of points as part of its data! Let's look at a few examples of processing this data structure. When the user clicks the mouse on the drawing surface, it's the start of a new curve, and a new *CurveData* object must be created and added to the list of curves. The instance variables in the new *CurveData* object must also be initialized. Here is the code from the `mousePressed()` routine that does this:

```
currentCurve = new CurveData();        // Create a new CurveData object.

currentCurve.color = currentColor;     // The color of the curve is taken from an
                                       // instance variable that represents the
                                       // currently selected drawing color.

currentCurve.symmetric = useSymmetry;  // The "symmetric" property of the curve
                                       // is also copied from the current value
                                       // of an instance variable, useSymmetry.

currentCurve.points = new ArrayList<Point>();  // Create a new point list object.

currentCurve.points.add( new Point(evt.getX(), evt.getY()) );
           // The point where the user pressed the mouse is the first point on
           // the curve.  A new Point object is created to hold the coordinates
           // of that point and is added to the list of points for the curve.

curves.add(currentCurve);   // Add the CurveData object to the list of curves.
```

As the user drags the mouse, new points are added to `currentCurve`, and `repaint()` is called. When the picture is redrawn, the new point will be part of the picture.

The `paintComponent()` method has to use the data in `curves` to draw all the curves. The basic structure is a for-each loop that processes the data for each individual curve in turn. This has the form:

```
for ( CurveData curve : curves ) {
   .
   .  // Draw the curve represented by the object, curve, of type CurveData.
   .
```

```
}
```

In the body of this loop, `curve.points` is a variable of type ArrayList<Point> that holds the list of points on the curve. The `i-th` point on the curve can be obtained by calling the `get()` method of this list: `curve.points.get(i)`. This returns a value of type *Point* which contains instance variables named `x` and `y`. We can refer directly to the x-coordinate of the `i-th` point as:

```
curve.points.get(i).x
```

This might seem rather complicated, but it's a nice example of a complex name that specifies a path to a desired piece of data: Go to the object, `curve`. Inside `curve`, go to `points`. Inside `points`, get the `i-th` item. And from that item, get the instance variable named `x`. Here is the complete definition of the `paintComponent()` method:

```
public void paintComponent(Graphics g) {
   super.paintComponent(g);
   for ( CurveData curve : curves) {
      g.setColor(curve.color);
      for (int i = 1; i < curve.points.size(); i++) {
            // Draw a line segment from point number i-1 to point number i.
         int x1 = curve.points.get(i-1).x;
         int y1 = curve.points.get(i-1).y;
         int x2 = curve.points.get(i).x;
         int y2 = curve.points.get(i).y;
         g.drawLine(x1,y1,x2,y2);
         if (curve.symmetric) {
               // Also draw the horizontal and vertical reflections
               // of the line segment.
            int w = getWidth();
            int h = getHeight();
            g.drawLine(w-x1,y1,w-x2,y2);
            g.drawLine(x1,h-y1,x2,h-y2);
            g.drawLine(w-x1,h-y1,w-x2,h-y2);
         }
      }
   }
} // end paintComponent()
```

I encourage you to read the full source code, *SimplePaint2.java*. In addition to serving as an example of using parameterized types, it also serves as another example of creating and using menus.

7.3.5 Vectors

The *ArrayList* class was introduced in Java version 1.2, as one of a group of classes designed for working with collections of objects. We'll look at these "collection classes" in Chapter 10. Early versions of Java did not include *ArrayList*, but they did have a very similar class named `java.util.Vector`. You can still see *Vectors* used in older code and in many of Java's standard classes, so it's worth knowing about them. Using a *Vector* is similar to using an *ArrayList*, except that different names are used for some commonly used instance methods, and some instance methods in one class don't correspond to any instance method in the other class.

Like an *ArrayList*, a *Vector* is similar to an array of *Objects* that can grow to be as large as necessary. The default constructor, `new Vector()`, creates a vector with no elements. Suppose that `vec` is a *Vector*. Then we have:

- `vec.size()` — a function that returns the number of elements currently in the vector.
- `vec.elementAt(N)` — returns the `N-th` element of the vector, for an integer `N`. `N` must be in the range 0 to `vec.size()-1`. This is the same as `get(N)` for an *ArrayList*.
- `vec.setElementAt(obj,N)` — sets the `N-th` element in the vector to be `obj`. `N` must be in the range 0 to `vec.size()-1`. This is the same as `set(N,obj)` for an *ArrayList*.
- `vec.addElement(obj)` — adds the *Object*, `obj`, to the end of the vector. This is the same as the `add()` method of an *ArrayList*.
- `vec.removeElement(obj)` — removes `obj` from the vector, if it occurs. Only the first occurrence is removed. This is the same as `remove(obj)` for an *ArrayList*.
- `vec.removeElementAt(N)` — removes the `N-th` element, for an integer `N`. `N` must be in the range 0 to `vec.size()-1`. This is the same as `remove(N)` for an *ArrayList*.
- `vec.setSize(N)` — sets the size of the vector to `N`. If there were more than `N` elements in `vec`, the extra elements are removed. If there were fewer than `N` elements, extra spaces are filled with `null`. The *ArrayList* class, unfortunately, does not have a `setSize()` method.

The *Vector* class includes many more methods, but these are probably the most commonly used. Note that in Java 5.0, *Vector* can be used as a parameterized type in exactly the same way as *ArrayList*. That is, if *BaseType* is any class or interface name, then Vector<BaseType> represents vectors that can hold only objects of type *BaseType*.

7.4 Searching and Sorting

TWO ARRAY PROCESSING TECHNIQUES that are particularly common are ***searching*** and ***sorting***. Searching here refers to finding an item in the array that meets some specified criterion. Sorting refers to rearranging all the items in the array into increasing or decreasing order (where the meaning of increasing and decreasing can depend on the context).

Sorting and searching are often discussed, in a theoretical sort of way, using an array of numbers as an example. In practical situations, though, more interesting types of data are usually involved. For example, the array might be a mailing list, and each element of the array might be an object containing a name and address. Given the name of a person, you might want to look up that person's address. This is an example of searching, since you want to find the object in the array that contains the given name. It would also be useful to be able to sort the array according to various criteria. One example of sorting would be ordering the elements of the array so that the names are in alphabetical order. Another example would be to order the elements of the array according to zip code before printing a set of mailing labels. (This kind of sorting can get you a cheaper postage rate on a large mailing.)

This example can be generalized to a more abstract situation in which we have an array that contains objects, and we want to search or sort the array based on the value of one of the instance variables in that array. We can use some terminology here that originated in work with "databases," which are just large, organized collections of data. We refer to each of the objects in the array as a ***record***. The instance variables in an object are then called ***fields*** of the record. In the mailing list example, each record would contain a name and address. The fields of the record might be the first name, last name, street address, state, city and zip code.

For the purpose of searching or sorting, one of the fields is designated to be the ***key*** field. Searching then means finding a record in the array that has a specified value in its key field. Sorting means moving the records around in the array so that the key fields of the record are in increasing (or decreasing) order.

In this section, most of my examples follow the tradition of using arrays of numbers. But I'll also give a few examples using records and keys, to remind you of the more practical applications.

7.4.1 Searching

There is an obvious algorithm for searching for a particular item in an array: Look at each item in the array in turn, and check whether that item is the one you are looking for. If so, the search is finished. If you look at every item without finding the one you want, then you can be sure that the item is not in the array. It's easy to write a subroutine to implement this algorithm. Let's say the array that you want to search is an array of **ints**. Here is a method that will search the array for a specified integer. If the integer is found, the method returns the index of the location in the array where it is found. If the integer is not in the array, the method returns the value `-1` as a signal that the integer could not be found:

```
/**
 * Searches the array A for the integer N.  If N is not in the array,
 * then -1 is returned.  If N is in the array, then the return value is
 * the first integer i that satisfies A[i] == N.
 */
static int find(int[] A, int N) {

   for (int index = 0; index < A.length; index++) {
      if ( A[index] == N )
         return index;  // N has been found at this index!
   }

   // If we get this far, then N has not been found
   // anywhere in the array.  Return a value of -1.

   return -1;

}
```

This method of searching an array by looking at each item in turn is called ***linear search***. If nothing is known about the order of the items in the array, then there is really no better alternative algorithm. But if the elements in the array are known to be in increasing or decreasing order, then a much faster search algorithm can be used. An array in which the elements are in order is said to be ***sorted***. Of course, it takes some work to sort an array, but if the array is to be searched many times, then the work done in sorting it can really pay off.

Binary search is a method for searching for a given item in a **sorted** array. Although the implementation is not trivial, the basic idea is simple: If you are searching for an item in a sorted list, then it is possible to eliminate half of the items in the list by inspecting a single item. For example, suppose that you are looking for the number 42 in a sorted array of 1000 integers. Let's assume that the array is sorted into increasing order. Suppose you check item number 500 in the array, and find that the item is 93. Since 42 is less than 93, and since the elements in the array are in increasing order, we can conclude that if 42 occurs in the array at all, then it must occur somewhere before location 500. All the locations numbered 500 or

above contain values that are greater than or equal to 93. These locations can be eliminated as possible locations of the number 42.

The next obvious step is to check location 250. If the number at that location is, say, -21, then you can eliminate locations before 250 and limit further search to locations between 251 and 499. The next test will limit the search to about 125 locations, and the one after that to about 62. After just 10 steps, there is only one location left. This is a whole lot better than looking through every element in the array. If there were a million items, it would still take only 20 steps for binary search to search the array! (Mathematically, the number of steps is approximately equal to the logarithm, in the base 2, of the number of items in the array.)

In order to make binary search into a Java subroutine that searches an array A for an item `N`, we just have to keep track of the range of locations that could possibly contain `N`. At each step, as we eliminate possibilities, we reduce the size of this range. The basic operation is to look at the item in the middle of the range. If this item is greater than `N`, then the second half of the range can be eliminated. If it is less than `N`, then the first half of the range can be eliminated. If the number in the middle just happens to be `N` exactly, then the search is finished. If the size of the range decreases to zero, then the number `N` does not occur in the array. Here is a subroutine that returns the location of `N` in a sorted array `A`. If `N` cannot be found in the array, then a value of -1 is returned instead:

```
/**
 * Searches the array A for the integer N.
 * Precondition:  A must be sorted into increasing order.
 * Postcondition: If N is in the array, then the return value, i,
 *    satisfies A[i] == N.  If N is not in the array, then the
 *    return value is -1.
 */
static int binarySearch(int[] A, int N) {

    int lowestPossibleLoc = 0;
    int highestPossibleLoc = A.length - 1;

    while (highestPossibleLoc >= lowestPossibleLoc) {
       int middle = (lowestPossibleLoc + highestPossibleLoc) / 2;
       if (A[middle] == N) {
                 // N has been found at this index!
          return middle;
       }
       else if (A[middle] > N) {
                 // eliminate locations >= middle
          highestPossibleLoc = middle - 1;
       }
       else {
                 // eliminate locations <= middle
          lowestPossibleLoc = middle + 1;
       }
    }

    // At this point, highestPossibleLoc < LowestPossibleLoc,
    // which means that N is known to be not in the array.  Return
    // a -1 to indicate that N could not be found in the array.

    return -1;

}
```

7.4.2 Association Lists

One particularly common application of searching is with ***association lists***. The standard example of an association list is a dictionary. A dictionary associates definitions with words. Given a word, you can use the dictionary to look up its definition. We can think of the dictionary as being a list of ***pairs*** of the form `(w,d)`, where `w` is a word and `d` is its definition. A general association list is a list of pairs `(k,v)`, where `k` is some "key" value, and `v` is a value associated to that key. In general, we want to assume that no two pairs in the list have the same key. There are two basic operations on association lists: Given a key, `k`, find the value `v` associated with `k`, if any. And given a key, `k`, and a value `v`, add the pair `(k,v)` to the association list (replacing the pair, if any, that had the same key value). The two operations are usually called ***get*** and ***put***.

Association lists are very widely used in computer science. For example, a compiler has to keep track of the location in memory associated with each variable. It can do this with an association list in which each key is a variable name and the associated value is the address of that variable in memory. Another example would be a mailing list, if we think of it as associating an address to each name on the list. As a related example, consider a phone directory that associates a phone number to each name. We'll look at a highly simplified version of this example. And note that things can be done much more efficiently, as you'll learn in Chapter 10.

The items in the phone directory's association list could be objects belonging to the class:

```
class PhoneEntry {
   String name;
   String phoneNum;
}
```

The data for a phone directory consists of an array of type PhoneEntry[] and an integer variable to keep track of how many entries are actually stored in the directory. The technique of "dynamic arrays" (Subsection 7.3.2) can be used in order to avoid putting an arbitrary limit on the number of entries that the phone directory can hold. Using an *ArrayList* would be another possibility. A *PhoneDirectory* class should include instance methods that implement the "get" and "put" operations. Here is one possible simple definition of the class:

```
/**
 * A PhoneDirectory holds a list of names with a phone number for
 * each name.  It is possible to find the number associated with
 * a given name, and to specify the phone number for a given name.
 */
public class PhoneDirectory {

   /**
    * An object of type PhoneEntry holds one name/number pair.
    */
   private static class PhoneEntry {
      String name;     // The name.
      String number;   // The associated phone number.
   }

   private PhoneEntry[] data;  // Array that holds the name/number pairs.
   private int dataCount;      // The number of pairs stored in the array.

   /**
```

```
 * Constructor creates an initially empty directory.
 */
public PhoneDirectory() {
   data = new PhoneEntry[1];
   dataCount = 0;
}

/**
 * Looks for a name/number pair with a given name.  If found, the index
 * of the pair in the data array is returned.  If no pair contains the
 * given name, then the return value is -1.  This private method is
 * used internally in getNumber() and putNumber().
 */
private int find( String name ) {
   for (int i = 0; i < dataCount; i++) {
      if (data[i].name.equals(name))
         return i;  // The name has been found in position i.
   }
   return -1;  // The name does not exist in the array.
}

/**
 * Finds the phone number, if any, for a given name.
 * @return The phone number associated with the name; if the name does
 *    not occur in the phone directory, then the return value is null.
 */
public String getNumber( String name ) {
   int position = find(name);
   if (position == -1)
      return null;   // There is no phone entry for the given name.
   else
      return data[position].number;
}

/**
 * Associates a given name with a given phone number.  If the name
 * already exists in the phone directory, then the new number replaces
 * the old one.  Otherwise, a new name/number pair is added.  The
 * name and number should both be non-null.  An IllegalArgumentException
 * is thrown if this is not the case.
 */
public void putNumber( String name, String number ) {
   if (name == null || number == null)
      throw new IllegalArgumentException("name and number cannot be null");
   int i = find(name);
   if (i >= 0) {
         // The name already exists, in position i in the array.
         // Just replace the old number at that position with the new.
      data[i].number = number;
   }
   else {
         // Add a new name/number pair to the array.  If the array is
         // already full, first create a new, larger array.
      if (dataCount == data.length) {
         PhoneEntry[] newData = new PhoneEntry[ 2*data.length ];
```

```
            System.arraycopy(data,0,newData,0,dataCount);
            data = newData;
         }
         PhoneEntry newEntry = new PhoneEntry();  // Create a new pair.
         newEntry.name = name;
         newEntry.number = number;
         data[dataCount] = newEntry;   // Add the new pair to the array.
         dataCount++;
      }
   }

} // end class PhoneDirectory
```

The class defines a private instance method, `find()`, that uses linear search to find the position of a given name in the array of name/number pairs. The `find()` method is used both in the `getNumber()` method and in the `putNumber()` method. Note in particular that `putNumber(name,number)` has to check whether the name is in the phone directory. If so, it just changes the number in the existing entry; if not, it has to create a new phone entry and add it to the array.

This class could use a lot of improvement. For one thing, it would be nice to use binary search instead of simple linear search in the `getNumber` method. However, we could only do that if the list of PhoneEntries were sorted into alphabetical order according to name. In fact, it's really not all that hard to keep the list of entries in sorted order, as you'll see in the next subsection.

7.4.3 Insertion Sort

We've seen that there are good reasons for sorting arrays. There are many algorithms available for doing so. One of the easiest to understand is the ***insertion sort*** algorithm. This method is also applicable to the problem of **keeping** a list in sorted order as you add new items to the list. Let's consider that case first:

Suppose you have a sorted list and you want to add an item to that list. If you want to make sure that the modified list is still sorted, then the item must be inserted into the right location, with all the smaller items coming before it and all the bigger items after it. This will mean moving each of the bigger items up one space to make room for the new item.

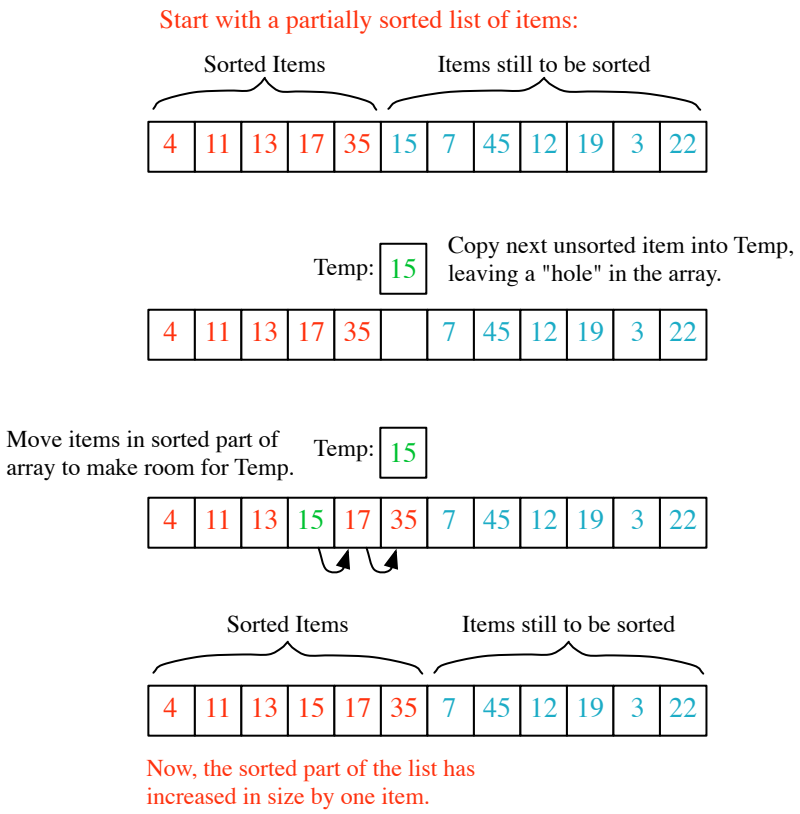
```
/*
 * Precondition:  itemsInArray is the number of items that are
 *    stored in A.  These items must be in increasing order
 *    (A[0] <= A[1] <= ... <= A[itemsInArray-1]).
 *    The array size is at least one greater than itemsInArray.
 * Postcondition:  The number of items has increased by one,
 *    newItem has been added to the array, and all the items
 *    in the array are still in increasing order.
 * Note:  To complete the process of inserting an item in the
 *    array, the variable that counts the number of items
 *    in the array must be incremented, after calling this
 *    subroutine.
 */
static void insert(int[] A, int itemsInArray, int newItem) {

   int loc = itemsInArray - 1;  // Start at the end of the array.
```

```
      /* Move items bigger than newItem up one space;
         Stop when a smaller item is encountered or when the
         beginning of the array (loc == 0) is reached. */

      while (loc >= 0 && A[loc] > newItem) {
         A[loc + 1] = A[loc];  // Bump item from A[loc] up to loc+1.
         loc = loc - 1;        // Go on to next location.
      }

      A[loc + 1] = newItem;  // Put newItem in last vacated space.

   }
```

Conceptually, this could be extended to a sorting method if we were to take all the items out of an unsorted array, and then insert them back into the array one-by-one, keeping the list in sorted order as we do so. Each insertion can be done using the `insert` routine given above. In the actual algorithm, we don't really take all the items from the array; we just remember what part of the array has been sorted:

```
static void insertionSort(int[] A) {
      // Sort the array A into increasing order.

   int itemsSorted; // Number of items that have been sorted so far.

   for (itemsSorted = 1; itemsSorted < A.length; itemsSorted++) {
         // Assume that items A[0], A[1], ... A[itemsSorted-1]
         // have already been sorted.  Insert A[itemsSorted]
         // into the sorted part of the list.

      int temp = A[itemsSorted];  // The item to be inserted.
      int loc = itemsSorted - 1;  // Start at end of list.

      while (loc >= 0 && A[loc] > temp) {
         A[loc + 1] = A[loc]; // Bump item from A[loc] up to loc+1.
         loc = loc - 1;       // Go on to next location.
      }

      A[loc + 1] = temp; // Put temp in last vacated space.
   }
}
```

The following is an illustration of one stage in insertion sort. It shows what happens during one execution of the `for` loop in the above method, when `itemsSorted` is 5:



7.4.4 Selection Sort

Another typical sorting method uses the idea of finding the biggest item in the list and moving it to the end—which is where it belongs if the list is to be in increasing order. Once the biggest item is in its correct location, you can then apply the same idea to the remaining items. That is, find the next-biggest item, and move it into the next-to-last space, and so forth. This algorithm is called ***selection sort***. It's easy to write:

```
static void selectionSort(int[] A) {
      // Sort A into increasing order, using selection sort

   for (int lastPlace = A.length-1; lastPlace > 0; lastPlace--) {
         // Find the largest item among A[0], A[1], ...,
         // A[lastPlace], and move it into position lastPlace
         // by swapping it with the number that is currently
         // in position lastPlace.

      int maxLoc = 0;  // Location of largest item seen so far.

      for (int j = 1; j <= lastPlace; j++) {
         if (A[j] > A[maxLoc]) {
               // Since A[j] is bigger than the maximum we've seen
               // so far, j is the new location of the maximum value
               // we've seen so far.
            maxLoc = j;
         }
```

```
        }

        int temp = A[maxLoc];  // Swap largest item with A[lastPlace].
        A[maxLoc] = A[lastPlace];
        A[lastPlace] = temp;

    }  // end of for loop

}
```

Insertion sort and selection sort are suitable for sorting fairly small arrays (up to a few hundred elements, say). There are more complicated sorting algorithms that are much faster than insertion sort and selection sort for large arrays. I'll discuss one such algorithm in Chapter 9.

* * *

A variation of selection sort is used in the *Hand* class that was introduced in Subsection 5.4.1. (By the way, you are finally in a position to fully understand the source code for both the *Hand* class and the *Deck* class from that section. See the source files *Deck.java* and *Hand.java*.)

In the *Hand* class, a hand of playing cards is represented by an *ArrayList*. The objects stored in the *ArrayList* are of type *Card*. A *Card* object contains instance methods `getSuit()` and `getValue()` that can be used to determine the suit and value of the card. In my sorting method, I actually create a new list and move the cards one-by-one from the old list to the new list. The cards are selected from the old list in increasing order. In the end, the new list becomes the hand and the old list is discarded. This is certainly not the most efficient procedure! But hands of cards are so small that the inefficiency is negligible. Here is the code for sorting cards by suit:

```
/**
 * Sorts the cards in the hand so that cards of the same suit are
 * grouped together, and within a suit the cards are sorted by value.
 * Note that aces are considered to have the lowest value, 1.
 */
public void sortBySuit() {
   ArrayList newHand = new ArrayList();
   while (hand.size() > 0) {
      int pos = 0;  // Position of minimal card.
      Card c = (Card)hand.get(0);  // Minimal card.
      for (int i = 1; i < hand.size(); i++) {
         Card c1 = (Card)hand.get(i);
         if ( c1.getSuit() < c.getSuit() ||
              (c1.getSuit() == c.getSuit() && c1.getValue() < c.getValue()) ) {
            pos = i;
            c = c1;
         }
      }
      hand.remove(pos);
      newHand.add(c);
   }
   hand = newHand;
}
```

This example illustrates the fact that comparing items in a list is not usually as simple as using the operator "<". In this case, we consider one card to be less than another if the suit of the first card is less than the suit of the second, and also if the suits are the same and the

value of the second card is less than the value of the first. The second part of this test ensures that cards with the same suit will end up sorted by value.

Sorting a list of *Strings* raises a similar problem: the "<" operator is not defined for strings. However, the *String* class does define a `compareTo` method. If `str1` and `str2` are of type *String*, then

```
str1.compareTo(str2)
```

returns an **int** that is 0 when `str1` is equal to `str2`, is less than 0 when `str1` precedes `str2`, and is greater than 0 when `str1` follows `str2`. The definition of "succeeds" and "follows" for strings uses what is called ***lexicographic ordering***, which is based on the Unicode values of the characters in the strings. Lexicographic ordering is not the same as alphabetical ordering, even for strings that consist entirely of letters (because in lexicographic ordering, all the upper case letters come before all the lower case letters). However, for words consisting strictly of the 26 lower case letters in the English alphabet, lexicographic and alphabetic ordering are the same. (The same holds true for uppercase letters.) Thus, if `str1` and `str2` are strings containing only letters from the English alphabet, then the test

```
str1.toLowerCase().compareTo(str2.toLowerCase()) < 0
```

is true if and only if `str1` comes before `str2` in alphabetical order.

7.4.5 Unsorting

I can't resist ending this section on sorting with a related problem that is much less common, but is a bit more fun. That is the problem of putting the elements of an array into a random order. The typical case of this problem is shuffling a deck of cards. A good algorithm for shuffling is similar to selection sort, except that instead of moving the biggest item to the end of the list, an item is selected at random and moved to the end of the list. Here is a subroutine to shuffle an array of `ints`:

```
/**
 * Postcondition:  The items in A have been rearranged into a random order.
 */
static void shuffle(int[] A) {
   for (int lastPlace = A.length-1; lastPlace > 0; lastPlace--) {
         // Choose a random location from among 0,1,...,lastPlace.
      int randLoc = (int)(Math.random()*(lastPlace+1));
         // Swap items in locations randLoc and lastPlace.
      int temp = A[randLoc];
      A[randLoc] = A[lastPlace];
      A[lastPlace] = temp;
   }
}
```

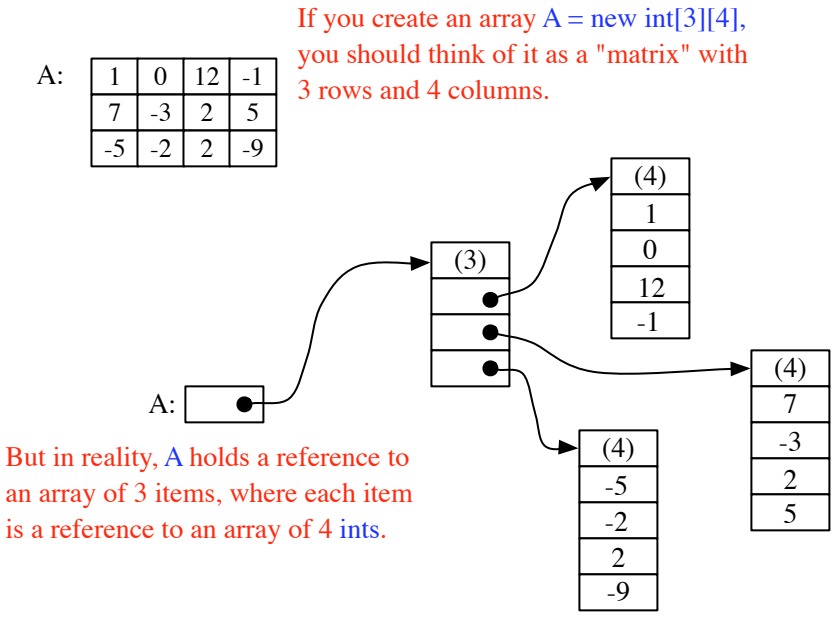
7.5 Multi-dimensional Arrays

Any type can be used as the base type of an array. You can have an array of **ints**, an array of *Strings*, an array of *Objects*, and so on. In particular, since an array type is a first-class Java type, you can have an array of arrays. For example, an array of **ints** has type int[]. This means that there is automatically another type, int[][], which represents an "array of arrays of ints". Such an array is said to be a ***two-dimensional array***. Of course once you have the

type int[][], there is nothing to stop you from forming the type int[][][], which represents a ***three-dimensional array***—and so on. There is no limit on the number of dimensions that an array type can have. However, arrays of dimension three or higher are fairly uncommon, and I concentrate here mainly on two-dimensional arrays. The type BaseType[][] is usually read "two-dimensional array of *BaseType*" or "*BaseType* array array".

7.5.1 Creating Two-dimensional Arrays

The declaration statement "`int[][] A;`" declares a variable named `A` of type int[][]. This variable can hold a reference to an object of type int[][]. The assignment statement "`A = new int[3][4];`" creates a new two-dimensional array object and sets `A` to point to the newly created object. As usual, the declaration and assignment could be combined in a single declaration statement "`int[][] A = new int[3][4];`". The newly created object is an array of arrays-of-**ints**. The notation `int[3][4]` indicates that there are 3 arrays-of-`ints` in the array `A`, and that there are 4 **ints** in each array-of-**ints**. However, trying to think in such terms can get a bit confusing—as you might have already noticed. So it is customary to think of a two-dimensional array of items as a rectangular ***grid*** or ***matrix*** of items. The notation "`new int[3][4]`" can then be taken to describe a grid of `ints` with 3 rows and 4 columns. The following picture might help:



For the most part, you can ignore the reality and keep the picture of a grid in mind. Sometimes, though, you will need to remember that each row in the grid is really an array in itself. These arrays can be referred to as `A[0]`, `A[1]`, and `A[2]`. Each row is in fact a value of type int[]. It could, for example, be passed to a subroutine that asks for a parameter of type int[].

The notation `A[1]` refers to one of the rows of the array `A`. Since `A[1]` is itself an array of **ints**, you can use another subscript to refer to one of the positions in that row. For example, `A[1][3]` refers to item number 3 in row number 1. Keep in mind, of course, that both rows and columns are numbered starting from zero. So, in the above example, `A[1][3]` is 5. More

generally, `A[i][j]` refers to the grid position in row number `i` and column number `j`. The 12 items in `A` are named as follows:

```
A[0][0]    A[0][1]    A[0][2]    A[0][3]
A[1][0]    A[1][1]    A[1][2]    A[1][3]
A[2][0]    A[2][1]    A[2][2]    A[2][3]
```

`A[i][j]` is actually a variable of type **int**. You can assign integer values to it or use it in any other context where an integer variable is allowed.

It might be worth noting that `A.length` gives the number of rows of `A`. To get the number of columns in `A`, you have to ask how many `ints` there are in a row; this number would be given by `A[0].length`, or equivalently by `A[1].length` or `A[2].length`. (There is actually no rule that says that all the rows of an array must have the same length, and some advanced applications of arrays use varying-sized rows. But if you use the `new` operator to create an array in the manner described above, you'll always get an array with equal-sized rows.)

Three-dimensional arrays are treated similarly. For example, a three-dimensional array of `ints` could be created with the declaration statement "`int[][][] B = new int[7][5][11];`". It's possible to visualize the value of `B` as a solid 7-by-5-by-11 block of cells. Each cell holds an **int** and represents one position in the three-dimensional array. Individual positions in the array can be referred to with variable names of the form `B[i][j][k]`. Higher-dimensional arrays follow the same pattern, although for dimensions greater than three, there is no easy way to visualize the structure of the array.

It's possible to fill a multi-dimensional array with specified items at the time it is declared. Recall that when an ordinary one-dimensional array variable is declared, it can be assigned an "array initializer," which is just a list of values enclosed between braces, `{` and `}`. Array initializers can also be used when a multi-dimensional array is declared. An initializer for a two-dimensional array consists of a list of one-dimensional array initializers, one for each row in the two-dimensional array. For example, the array `A` shown in the picture above could be created with:

```
int[][]  A  =  {  {  1,  0, 12, -1 },
                  {  7, -3,  2,  5 },
                  { -5, -2,  2, -9 }
               };
```

If no initializer is provided for an array, then when the array is created it is automatically filled with the appropriate value: zero for numbers, `false` for boolean, and `null` for objects.

7.5.2 Using Two-dimensional Arrays

Just as in the case of one-dimensional arrays, two-dimensional arrays are often processed using `for` statements. To process all the items in a two-dimensional array, you have to use one `for` statement nested inside another. If the array `A` is declared as

```
int[][]  A  =  new int[3][4];
```

then you could store a 17 into each location in `A` with:

```
for (int row = 0;  row < 3;  row++) {
   for (int column = 0;  column < 4;  column++) {
      A[row][column] = 17;
   }
}
```

The first time the outer `for` loop executes (with `row` = 0), the inner for loop fills in the four values in the first row of A, namely `A[0][0] = 17`, `A[0][1] = 17`, `A[0][2] = 17`, and `A[0][3] = 17`. The next execution of the outer `for` loop fills in the second row of `A`. And the third and final execution of the outer loop fills in the final row of `A`.

Similarly, you could add up all the items in `A` with:

```
int sum = 0;
for (int i = 0; i < 3; i++)
   for (int j = 0; j < 4; j++)
      sum = sum + A[i][j];
```

This could even be done with nested for-each loops. Keep in mind that the elements in `A` are objects of type int[], while the elements in each row of `A` are of type **int**:

```
int sum = 0;
for ( int[] row : A ) {       // For each row in A...
   for ( int item : row )     // For each item in that row...
      sum = sum + item;       // Add item to the sum.
}
```

To process a three-dimensional array, you would, of course, use triply nested `for` loops.

* * *

A two-dimensional array can be used whenever the data that is being represented can be arranged into rows and columns in a natural way. Often, the grid is built into the problem. For example, a chess board is a grid with 8 rows and 8 columns. If a class named *ChessPiece* is available to represent individual chess pieces, then the contents of a chess board could be represented by a two-dimensional array:

```
ChessPiece[][]  board  =  new ChessPiece[8][8];
```

Or consider the "mosaic" of colored rectangles used in an example in Subsection 4.6.2. The mosaic is implemented by a class named *MosaicCanvas.java*. The data about the color of each of the rectangles in the mosaic is stored in an instance variable named `grid` of type Color[][]. Each position in this grid is occupied by a value of type *Color*. There is one position in the grid for each colored rectangle in the mosaic. The actual two-dimensional array is created by the statement:

```
grid  =  new Color[ROWS][COLUMNS];
```

where `ROWS` is the number of rows of rectangles in the mosaic and `COLUMNS` is the number of columns. The value of the `Color` variable `grid[i][j]` is the color of the rectangle in row number `i` and column number `j`. When the color of that rectangle is changed to some color, `c`, the value stored in `grid[i][j]` is changed with a statement of the form "`grid[i][j] = c;`". When the mosaic is redrawn, the values stored in the two-dimensional array are used to decide what color to make each rectangle. Here is a simplified version of the code from the *MosaicCanvas* class that draws all the colored rectangles in the grid. You can see how it uses the array:

```
int rowHeight = getHeight() / ROWS;
int colWidth = getWidth() / COLUMNS;
for (int row = 0; row < ROWS; row++) {
   for (int col = 0; col < COLUMNS; col++) {
       g.setColor( grid[row][col] ); // Get color from array.
       g.fillRect( col*colWidth, row*rowHeight,
                                      colWidth, rowHeight );
   }
}
```

Sometimes two-dimensional arrays are used in problems in which the grid is not so visually obvious. Consider a company that owns 25 stores. Suppose that the company has data about the profit earned at each store for each month in the year 2010. If the stores are numbered from 0 to 24, and if the twelve months from January '10 through December '10 are numbered from 0 to 11, then the profit data could be stored in an array, `profit`, constructed as follows:

```
double[][]  profit  =  new double[25][12];
```

`profit[3][2]` would be the amount of profit earned at store number 3 in March, and more generally, `profit[storeNum][monthNum]` would be the amount of profit earned in store number `storeNum` in month number `monthNum`. In this example, the one-dimensional array `profit[storeNum]` has a very useful meaning: It is just the profit data for one particular store for all the months in the whole year.

Let's assume that the `profit` array has already been filled with data. This data can be processed in a lot of interesting ways. For example, the total profit for the company—for the whole year from all its stores—can be calculated by adding up all the entries in the array:

```
double totalProfit;  // Company's total profit in 2010.

totalProfit = 0;
for (int store = 0; store < 25; store++) {
   for (int month = 0; month < 12; month++)
      totalProfit += profit[store][month];
}
```

Sometimes it is necessary to process a single row or a single column of an array, not the entire array. For example, to compute the total profit earned by the company in December, that is, in month number 11, you could use the loop:

```
double decemberProfit = 0.0;
for (storeNum = 0; storeNum < 25; storeNum++)
   decemberProfit += profit[storeNum][11];
```

Let's extend this idea to create a one-dimensional array that contains the total profit for each month of the year:

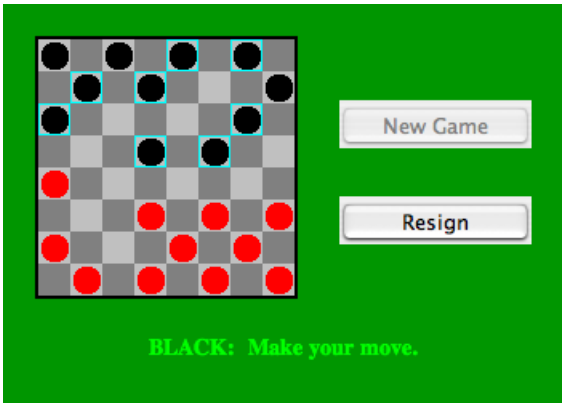
```
double[] monthlyProfit;  // Holds profit for each month.
monthlyProfit = new double[12];

for (int month = 0; month < 12; month++) {
   // compute the total profit from all stores in this month.
   monthlyProfit[month] = 0.0;
   for (int store = 0; store < 25; store++) {
         // Add the profit from this store in this month
         // into the total profit figure for the month.
      monthlyProfit[month] += profit[store][month];
   }
}
```

As a final example of processing the profit array, suppose that we wanted to know which store generated the most profit over the course of the year. To do this, we have to add up the monthly profits for each store. In array terms, this means that we want to find the sum of each row in the array. As we do this, we need to keep track of which row produces the largest total.

```
double maxProfit; // Maximum profit earned by a store.
int bestStore;    // The number of the store with the
                  //   maximum profit.

double total;    // Total profit for one store.

// First compute the profit from store number 0.

total = 0.0;
for (month = 0;  month < 12; month++)
   total += profit[0][month];

bestStore = 0;      // Start by assuming that the best
maxProfit = total;  //     store is store number 0.

// Now, go through the other stores, and whenever we
// find one with a bigger profit than maxProfit, revise
// the assumptions about bestStore and maxProfit.

for (store = 1; store < 25; store++) {

    // Compute this store's profit for the year.

    total = 0.0;
    for (month = 0; month < 12; month++)
       total += profit[store][month];

    // Compare this store's profits with the highest
    // profit we have seen among the preceding stores.

    if (total > maxProfit) {
       maxProfit = total;    // Best profit seen so far!
       bestStore = store;    // It came from this store.
    }

} // end for

// At this point, maxProfit is the best profit of any
// of the 25 stores, and bestStore is a store that
// generated that profit.  (Note that there could also be
// other stores that generated exactly the same profit.)
```

7.5.3 Example: Checkers

For the rest of this section, we'll look at a more substantial example. We look at a program that lets two users play checkers against each other. A player moves by clicking on the piece to be moved and then on the empty square to which it is to be moved. The squares that the current player can legally click are highlighted. The square containing a piece that has been selected to be moved is surrounded by a white border. Other pieces that can legally be moved are surrounded by a cyan-colored border. If a piece has been selected, each empty square that it can legally move to is highlighted with a green border. The game enforces the rule that if the current player can jump one of the opponent's pieces, then the player must jump. When a player's piece becomes a king, by reaching the opposite end of the board, a big white "K" is drawn on the piece. You can try an applet version of the program in the on-line version of this section. Here is what it looks like:



I will only cover a part of the programming of this applet. I encourage you to read the complete source code, *Checkers.java*. At over 750 lines, this is a more substantial example than anything you've seen before in this course, but it's an excellent example of state-based, event-driven programming.

The data about the pieces on the board are stored in a two-dimensional array. Because of the complexity of the program, I wanted to divide it into several classes. In addition to the main class, there are several nested classes. One of these classes is *CheckersData*, which handles the data for the board. It is mainly this class that I want to talk about.

The *CheckersData* class has an instance variable named `board` of type `int[][]`. The value of board is set to "`new int[8][8]`", an 8-by-8 grid of integers. The values stored in the grid are defined as constants representing the possible contents of a square on a checkerboard:

```
static final int
          EMPTY = 0,          // Value representing an empty square.
          RED = 1,            // A regular red piece.
          RED_KING = 2,       // A red king.
          BLACK = 3,          // A regular black piece.
          BLACK_KING = 4;     // A black king.
```

The constants `RED` and `BLACK` are also used in my program (or, perhaps, misused) to represent the two players in the game. When a game is started, the values in the variable, `board`, are set to represent the initial state of the board. The grid of values looks like

	0	1	2	3	4	5	6	7
0	BLACK	EMPTY	BLACK	EMPTY	BLACK	EMPTY	BLACK	EMPTY
1	EMPTY	BLACK	EMPTY	BLACK	EMPTY	BLACK	EMPTY	BLACK
2	BLACK	EMPTY	BLACK	EMPTY	BLACK	EMPTY	BLACK	EMPTY
3	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY
4	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY
5	EMPTY	RED	EMPTY	RED	EMPTY	RED	EMPTY	RED
6	RED	EMPTY	RED	EMPTY	RED	EMPTY	RED	EMPTY
7	EMPTY	RED	EMPTY	RED	EMPTY	RED	EMPTY	RED

A regular black piece can only move "down" the grid. That is, the row number of the square it moves to must be greater than the row number of the square it comes from. A regular red piece can only move up the grid. Kings of either color, of course, can move in both directions.

One function of the *CheckersData* class is to take care of all the details of making moves on the board. An instance method named `makeMove()` is provided to do this. When a player moves a piece from one square to another, the values stored at two positions in the array are changed. But that's not all. If the move is a jump, then the piece that was jumped is removed from the board. (The method checks whether the move is a jump by checking if the square to which the piece is moving is two rows away from the square where it starts.) Furthermore, a `RED` piece that moves to row 0 or a `BLACK` piece that moves to row 7 becomes a king. This is good programming: the rest of the program doesn't have to worry about any of these details. It just calls this `makeMove()` method:

```
/**
 * Make the move from (fromRow,fromCol) to (toRow,toCol).  It is
 * ASSUMED that this move is legal!  If the move is a jump, the
 * jumped piece is removed from the board.  If a piece moves
 * to the last row on the opponent's side of the board, the
 * piece becomes a king.
 */
void makeMove(int fromRow, int fromCol, int toRow, int toCol) {

   board[toRow][toCol] = board[fromRow][fromCol]; // Move the piece.
   board[fromRow][fromCol] = EMPTY;

   if (fromRow - toRow == 2 || fromRow - toRow == -2) {
         // The move is a jump.  Remove the jumped piece from the board.
      int jumpRow = (fromRow + toRow) / 2; // Row of the jumped piece.
      int jumpCol = (fromCol + toCol) / 2; // Column of the jumped piece.
      board[jumpRow][jumpCol] = EMPTY;
   }

   if (toRow == 0 && board[toRow][toCol] == RED)
      board[toRow][toCol] = RED_KING;  // Red piece becomes a king.
   if (toRow == 7 && board[toRow][toCol] == BLACK)
      board[toRow][toCol] = BLACK_KING;  // Black piece becomes a king.

}  // end makeMove()
```

An even more important function of the *CheckersData* class is to find legal moves on the board. In my program, a move in a Checkers game is represented by an object belonging to the following class:

```
/**
 * A CheckersMove object represents a move in the game of
 * Checkers.  It holds the row and column of the piece that is
 * to be moved and the row and column of the square to which
 * it is to be moved.  (This class makes no guarantee that
 * the move is legal.)
 */
private static class CheckersMove {

   int fromRow, fromCol;  // Position of piece to be moved.
   int toRow, toCol;      // Square it is to move to.

   CheckersMove(int r1, int c1, int r2, int c2) {
```

```
        // Constructor.  Set the values of the instance variables.
      fromRow = r1;
      fromCol = c1;
      toRow = r2;
      toCol = c2;
   }

   boolean isJump() {
        // Test whether this move is a jump.  It is assumed that
        // the move is legal.  In a jump, the piece moves two
        // rows.  (In a regular move, it only moves one row.)
      return (fromRow - toRow == 2 || fromRow - toRow == -2);
   }

}  // end class CheckersMove.
```

The *CheckersData* class has an instance method which finds all the legal moves that are currently available for a specified player. This method is a function that returns an array of type CheckersMove[]. The array contains all the legal moves, represented as *CheckersMove* objects. The specification for this method reads

```
/**
 * Return an array containing all the legal CheckersMoves
 * for the specified player on the current board.  If the player
 * has no legal moves, null is returned.  The value of player
 * should be one of the constants RED or BLACK; if not, null
 * is returned.  If the returned value is non-null, it consists
 * entirely of jump moves or entirely of regular moves, since
 * if the player can jump, only jumps are legal moves.
 */
CheckersMove[] getLegalMoves(int player)
```

A brief pseudocode algorithm for the method is

```
Start with an empty list of moves
Find any legal jumps and add them to the list
if there are no jumps:
   Find any other legal moves and add them to the list
if the list is empty:
   return null
else:
   return the list
```

Now, what is this "list"? We have to return the legal moves in an array. But since an array has a fixed size, we can't create the array until we know how many moves there are, and we don't know that until near the end of the method, after we've already made the list! A neat solution is to use an *ArrayList* instead of an array to hold the moves as we find them. In fact, I use an object defined by the parameterized type ArrayList<CheckersMove> so that the list is restricted to holding objects of type *CheckersMove*. As we add moves to the list, it will grow just as large as necessary. At the end of the method, we can create the array that we really want and copy the data into it:

```
Let "moves" be an empty ArrayList<CheckersMove>
Find any legal jumps and add them to moves
if moves.size() is 0:
   Find any other legal moves and add them to moves
```

```
if moves.size() is 0:
   return null
else:
   Let moveArray be an array of CheckersMoves of length moves.size()
   Copy the contents of moves into moveArray
   return moveArray
```

Now, how do we find the legal jumps or the legal moves? The information we need is in the `board` array, but it takes some work to extract it. We have to look through all the positions in the array and find the pieces that belong to the current player. For each piece, we have to check each square that it could conceivably move to, and check whether that would be a legal move. If we are looking for legal jumps, we want to look at squares that are two rows and two columns away from the piece. There are four squares to consider. Thus, the line in the algorithm that says "Find any legal jumps and add them to moves" expands to:

```
For each row of the board:
   For each column of the board:
      if one of the player's pieces is at this location:
         if it is legal to jump to row + 2, column + 2
             add this move to moves
         if it is legal to jump to row - 2, column + 2
             add this move to moves
         if it is legal to jump to row + 2, column - 2
             add this move to moves
         if it is legal to jump to row - 2, column - 2
             add this move to moves
```

The line that says "Find any other legal moves and add them to moves" expands to something similar, except that we have to look at the four squares that are one column and one row away from the piece. Testing whether a player can legally move from one given square to another given square is itself non-trivial. The square the player is moving to must actually be on the board, and it must be empty. Furthermore, regular red and black pieces can only move in one direction. I wrote the following utility method to check whether a player can make a given non-jump move:

```
/**
 * This is called by the getLegalMoves() method to determine
 * whether the player can legally move from (r1,c1) to (r2,c2).
 * It is ASSUMED that (r1,c1) contains one of the player's
 * pieces and that (r2,c2) is a neighboring square.
 */
private boolean canMove(int player, int r1, int c1, int r2, int c2) {

   if (r2 < 0 || r2 >= 8 || c2 < 0 || c2 >= 8)
      return false;  // (r2,c2) is off the board.

   if (board[r2][c2] != EMPTY)
      return false;  // (r2,c2) already contains a piece.

   if (player == RED) {
      if (board[r1][c1] == RED && r2 > r1)
          return false;  // Regular red piece can only move down.
       return true;  // The move is legal.
   }
   else {
```

```
      if (board[r1][c1] == BLACK && r2 < r1)
          return false;  // Regular black piece can only move up.
       return true;  // The move is legal.
   }

}  // end canMove()
```

This method is called by my `getLegalMoves()` method to check whether one of the possible moves that it has found is actually legal. I have a similar method that is called to check whether a jump is legal. In this case, I pass to the method the square containing the player's piece, the square that the player might move to, and the square between those two, which the player would be jumping over. The square that is being jumped must contain one of the opponent's pieces. This method has the specification:

```
/**
 * This is called by other methods to check whether
 * the player can legally jump from (r1,c1) to (r3,c3).
 * It is assumed that the player has a piece at (r1,c1), that
 * (r3,c3) is a position that is 2 rows and 2 columns distant
 * from (r1,c1) and that (r2,c2) is the square between (r1,c1)
 * and (r3,c3).
 */
private boolean canJump(int player, int r1, int c1,
                                    int r2, int c2, int r3, int c3) { . . .
```

Given all this, you should be in a position to understand the complete `getLegalMoves()` method. It's a nice way to finish off this chapter, since it combines several topics that we've looked at: one-dimensional arrays, *ArrayLists*, and two-dimensional arrays:

```
CheckersMove[] getLegalMoves(int player) {

   if (player != RED && player != BLACK)
      return null;

   int playerKing;  // The constant for a King belonging to the player.
   if (player == RED)
      playerKing = RED_KING;
   else
      playerKing = BLACK_KING;

   ArrayList<CheckersMove> moves = new ArrayList<CheckersMove>();
               // Moves will be stored in this list.

   /*  First, check for any possible jumps.  Look at each square on
       the board.  If that square contains one of the player's pieces,
       look at a possible jump in each of the four directions from that
       square.  If there is a legal jump in that direction, put it in
       the moves ArrayList.
   */

   for (int row = 0; row < 8; row++) {
      for (int col = 0; col < 8; col++) {
        if (board[row][col] == player || board[row][col] == playerKing) {
            if (canJump(player, row, col, row+1, col+1, row+2, col+2))
               moves.add(new CheckersMove(row, col, row+2, col+2));
            if (canJump(player, row, col, row-1, col+1, row-2, col+2))
               moves.add(new CheckersMove(row, col, row-2, col+2));
```

```
            if (canJump(player, row, col, row+1, col-1, row+2, col-2))
               moves.add(new CheckersMove(row, col, row+2, col-2));
            if (canJump(player, row, col, row-1, col-1, row-2, col-2))
               moves.add(new CheckersMove(row, col, row-2, col-2));
         }
      }
   }

   /*  If any jump moves were found, then the user must jump, so we
       don't add any regular moves.  However, if no jumps were found,
       check for any legal regular moves.  Look at each square on
       the board.  If that square contains one of the player's pieces,
       look at a possible move in each of the four directions from
       that square.  If there is a legal move in that direction,
       put it in the moves ArrayList.
   */

   if (moves.size() == 0) {
      for (int row = 0; row < 8; row++) {
         for (int col = 0; col < 8; col++) {
           if (board[row][col] == player || board[row][col] == playerKing) {
              if (canMove(player,row,col,row+1,col+1))
                 moves.add(new CheckersMove(row,col,row+1,col+1));
              if (canMove(player,row,col,row-1,col+1))
                 moves.add(new CheckersMove(row,col,row-1,col+1));
              if (canMove(player,row,col,row+1,col-1))
                 moves.add(new CheckersMove(row,col,row+1,col-1));
              if (canMove(player,row,col,row-1,col-1))
                 moves.add(new CheckersMove(row,col,row-1,col-1));
           }
         }
      }
   }

   /* If no legal moves have been found, return null.  Otherwise, create
      an array just big enough to hold all the legal moves, copy the
      legal moves from the ArrayList into the array, and return the array.
   */

   if (moves.size() == 0)
      return null;
   else {
      CheckersMove[] moveArray = new CheckersMove[moves.size()];
      for (int i = 0; i < moves.size(); i++)
         moveArray[i] = moves.get(i);
      return moveArray;
   }

}  // end getLegalMoves
```

Exercises for Chapter 7

1. An example in Subsection 7.2.4 tried to answer the question, How many random people do you have to select before you find a duplicate birthday? The source code for that program can be found in the file *BirthdayProblemDemo.java*. Here are some related questions:

- How many random people do you have to select before you find **three** people who share the same birthday? (That is, all three people were born on the same day in the same month, but not necessarily in the same year.)
- Suppose you choose 365 people at random. How many different birthdays will they have? (The number could theoretically be anywhere from 1 to 365).
- How many different people do you have to check before you've found at least one person with a birthday on each of the 365 days of the year?

Write **three** programs to answer these questions. Each of your programs should simulate choosing people at random and checking their birthdays. (In each case, ignore the possibility of leap years.)

2. Write a program that will read a sequence of positive real numbers entered by the user and will print the same numbers in sorted order from smallest to largest. The user will input a zero to mark the end of the input. Assume that at most 100 positive numbers will be entered.

3. A ***polygon*** is a geometric figure made up of a sequence of connected line segments. The points where the line segments meet are called the ***vertices*** of the polygon. The *Graphics* class includes commands for drawing and filling polygons. For these commands, the coordinates of the vertices of the polygon are stored in arrays. If `g` is a variable of type *Graphics* then

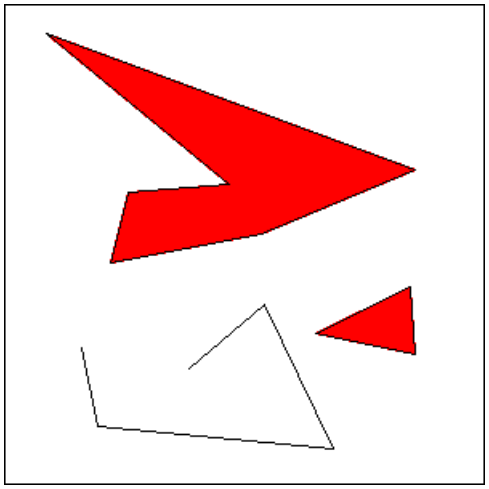
- `g.drawPolygon(xCoords, yCoords, pointCt)` will draw the outline of the polygon with vertices at the points `(xCoords[0],yCoords[0])`, `(xCoords[1],yCoords[1])`, ..., `(xCoords[pointCt-1],yCoords[pointCt-1])`. The third parameter, `pointCt`, is an **int** that specifies the number of vertices of the polygon. Its value should be 3 or greater. The first two parameters are arrays of type `int[]`. Note that the polygon automatically includes a line from the last point, `(xCoords[pointCt-1],yCoords[pointCt-1])`, back to the starting point `(xCoords[0],yCoords[0])`.
- `g.fillPolygon(xCoords, yCoords, pointCt)` fills the interior of the polygon with the current drawing color. The parameters have the same meaning as in the `drawPolygon()` method. Note that it is OK for the sides of the polygon to cross each other, but the interior of a polygon with self-intersections might not be exactly what you expect.

Write a panel class that lets the user draw polygons, and use your panel as the content pane in an applet (or standalone application). As the user clicks a sequence of points, count them and store their x- and y-coordinates in two arrays. These points will be the vertices of the polygon. Also, draw a line between each consecutive pair of points to give the user some visual feedback. When the user clicks near the starting point, draw the

complete polygon. Draw it with a red interior and a black border. The user should then be able to start drawing a new polygon. When the user shift-clicks on the applet, clear it.

For this exercise, there is no need to store information about the contents of the applet. Do the drawing directly in the `mousePressed()` routine, and use the `getGraphics()` method to get a *Graphics* object that you can use to draw the line. (Remember, though, that this is considered to be bad style.) You will not need a `paintComponent()` method, since the default action of filling the panel with its background color is good enough.

Here is a picture of my solution after the user has drawn a few polygons:



4. For this problem, you will need to use an array of objects. The objects belong to the class *MovingBall*, which I have already written. You can find the source code for this class in the file *MovingBall.java*. A *MovingBall* represents a circle that has an associated color, radius, direction, and speed. It is restricted to moving inside some rectangle in the `(x,y)` plane. It will "bounce back" when it hits one of the sides of this rectangle. A *MovingBall* does not actually move by itself. It's just a collection of data. You have to call instance methods to tell it to update its position and to draw itself. The constructor for the *MovingBall* class takes the form

```
new MovingBall(xmin, xmax, ymin, ymax)
```

where the parameters are integers that specify the limits on the `x` and `y` coordinates of the ball. (This sets the rectangle inside which the ball will stay.) In this exercise, you will want balls to bounce off the sides of the applet, so you will create them with the constructor call

```
new MovingBall(0, getWidth(), 0, getHeight())
```

The constructor creates a ball that initially is colored red, has a radius of 5 pixels, is located at the center of its range, has a random speed between 4 and 12, and is headed in a random direction. There is one **problem** here: You can't use this constructor until the width and height of the component are known. It would be OK to use it in the `init()` method of an applet, but not in the constructor of an applet or panel class. If you are using a panel class to display the ball, one slightly messy solution is to create the *MovingBall* objects in the panel's `paintComponent()` method the first time that method is called. You

can be sure that the size of the panel has been determined before `paintComponent()` is called. This is what I did in my own solution to this exercise.

If `ball` is a variable of type *MovingBall*, then the following methods are available:

- `ball.draw(g)` — draw the ball in a graphics context. The parameter, `g`, must be of type `Graphics`. (The drawing color in `g` will be changed to the color of the ball.)
- `ball.travel()` — change the `(x,y)`-coordinates of the ball by an amount equal to its speed. The ball has a certain direction of motion, and the ball is moved in that direction. Ordinarily, you will call this once for each frame of an animation, so the speed is given in terms of "pixels per frame". Calling this routine does not move the ball on the screen. It just changes the values of some instance variables in the object. The next time the object's `draw()` method is called, the ball will be drawn in the new position.
- `ball.headTowards(x,y)` — change the direction of motion of the ball so that it is headed towards the point `(x,y)`. This does not affect the speed.

These are the methods that you will need for this exercise. There are also methods for setting various properties of the ball, such as `ball.setColor(color)` for changing the color and `ball.setRadius(radius)` for changing its size. See the source code for more information. A nice variation on the exercise would be to use random colors and sizes for the balls.

For this exercise, you should create an applet that shows an animation of balls bouncing around on a black background. Use a *Timer* to drive the animation. (See Subsection 6.5.1.) Use an array of type `MovingBall[]` to hold the data for the balls. In addition, your program should listen for mouse and mouse motion events. When the user presses the mouse or drags the mouse, call each of the ball's `headTowards()` methods to make the balls head towards the mouse's location. My solution uses 50 balls and a time delay of 50 milliseconds for the timer.

5. The sample program *RandomArtPanel.java* from Subsection 6.5.1 shows a different random "artwork" every four seconds. There are three types of "art", one made from lines, one from circles, and one from filled squares. However, the program does not save the data for the picture that is shown on the screen. As a result, the picture cannot be redrawn when necessary. In fact, every time `paintComponent()` is called, a new picture is drawn.

Write a new version of *RandomArtPanel.java* that saves the data needed to redraw its pictures. The `paintComponent()` method should simply use the data to draw the picture. New data should be recomputed only every four seconds, in response to an event from the timer that drives the program.

To make this interesting, write a separate class for each of the three different types of art. Also write an abstract class to serve as the common base class for the three classes. Since all three types of art use a random gray background, the background color can be defined in their superclass. The superclass also contains a `draw()` method that draws the picture; this is an abstract method because its implementation depends on the particular type of art that is being drawn. The abstract class can be defined as:

```
private abstract class ArtData {
   Color backgroundColor;  // The background color for the art.
   ArtData() {  // Constructor sets background color to be a random gray.
      int x = (int)(256*Math.random());
```

```
            backgroundColor = new Color( x, x, x );
        }
        abstract void draw(Graphics g);  // Draws this artwork.
    }
```

Each of the three subclasses of `ArtData` must define its own `draw()` method. It must also define instance variables to hold the data necessary to draw the picture. I suggest that you should create random data for the picture in the constructor of the class, so that constructing the object will automatically create the data for the random artwork. (One problem with this is that you can't create the data until you know the size of the panel, so you can't create an `ArtData` object in the constructor of the panel. One solution is to create an `ArtData` object at the beginning of the `paintComponent()` method, if the object has not already been created.) In all three subclasses, you will need to use several arrays to store the data.

The file *RandomArtPanel.java* only defines a panel class. A main program that uses this panel can be found in *RandomArt.java*, and an applet that uses it can be found in *RandomArtApplet.java*. You only need to modify *RandomArtPanel*.

6. Write a program that will read a text file selected by the user, and will make an alphabetical list of all the different words in that file. All words should be converted to lower case, and duplicates should be eliminated from the list. The list should be written to an output file selected by the user. As discussed in Subsection 2.4.5, you can use *TextIO* to read and write files. Use a variable of type ArrayList<String> to store the words. (See Subsection 7.3.4.) It is not easy to separate a file into words as you are reading it. You can use the following method:

```
/**
 * Read the next word from TextIO, if there is one.  First, skip past
 * any non-letters in the input.  If an end-of-file is encountered before
 * a word is found, return null.  Otherwise, read and return the word.
 * A word is defined as a sequence of letters.  Also, a word can include
 * an apostrophe if the apostrophe is surrounded by letters on each side.
 * @return the next word from TextIO, or null if an end-of-file is
 * encountered
 */
private static String readNextWord() {
   char ch = TextIO.peek(); // Look at next character in input.
   while (ch != TextIO.EOF && ! Character.isLetter(ch)) {
          // Skip past non-letters.
      TextIO.getAnyChar();  // Read the character.
      ch = TextIO.peek();   // Look at the next character.
   }
   if (ch == TextIO.EOF) // Encountered end-of-file
      return null;
   // At this point, we know the next character is a letter, so read a word.
   String word = "";  // This will be the word that is read.
   while (true) {
      word += TextIO.getAnyChar();  // Append the letter onto word.
      ch = TextIO.peek();  // Look at next character.
      if ( ch == '\'' ) {
            // The next character is an apostrophe.  Read it, and
            // if the following character is a letter, add both the
```

```
                // apostrophe and the letter onto the word and continue
                // reading the word.  If the character after the apostrophe
                // is not a letter, the word is done, so break out of the loop.
             TextIO.getAnyChar();   // Read the apostrophe.
             ch = TextIO.peek();    // Look at char that follows apostrophe.
             if (Character.isLetter(ch)) {
                word += "\'" + TextIO.getAnyChar();
                ch = TextIO.peek();  // Look at next char.
             }
             else
                break;
          }
          if ( ! Character.isLetter(ch) ) {
                // If the next character is not a letter, the word is
                // finished, so break out of the loop.
             break;
          }
          // If we haven't broken out of the loop, next char is a letter.
       }
       return word;  // Return the word that has been read.
    }
```

Note that this method will return `null` when the file has been entirely read. You can use this as a signal to stop processing the input file.

7. The game of Go Moku (also known as Pente or Five Stones) is similar to Tic-Tac-Toe, except that it is played on a much larger board and the object is to get five squares in a row rather than three. Players take turns placing pieces on a board. A piece can be placed in any empty square. The first player to get five pieces in a row—horizontally, vertically, or diagonally—wins. If all squares are filled before either player wins, then the game is a draw. Write a program that lets two players play Go Moku against each other.

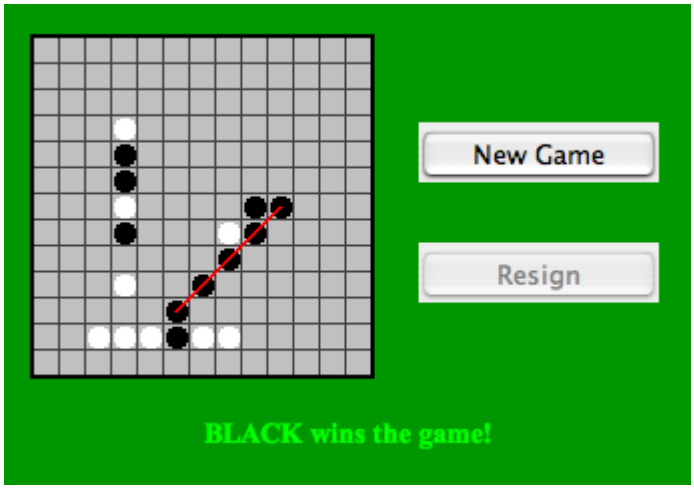
Your program will be simpler than the *Checkers* program from Subsection 7.5.3. Play alternates strictly between the two players, and there is no need to highlight the legal moves. You will only need two classes, a short panel class to set up the interface and a *Board* class to draw the board and do all the work of the game. Nevertheless, you will probably want to look at the source code for the checkers program, *Checkers.java*, for ideas about the general outline of the program.

The hardest part of the program is checking whether the move that a player makes is a winning move. To do this, you have to look in each of the four possible directions from the square where the user has placed a piece. You have to count how many pieces that player has in a row in that direction. If the number is five or more in any direction, then that player wins. As a hint, here is part of the code from my applet. This code counts the number of pieces that the user has in a row in a specified direction. The direction is specified by two integers, `dirX` and `dirY`. The values of these variables are 0, 1, or -1, and at least one of them is non-zero. For example, to look in the horizontal direction, `dirX` is 1 and `dirY` is 0.

```
int ct = 1;  // Number of pieces in a row belonging to the player.

int r, c;    // A row and column to be examined

r = row + dirX;  // Look at square in specified direction.
```

```
c = col + dirY;
while ( r >= 0 && r < 13 && c >= 0 && c < 13
                                    && board[r][c] == player ) {
        // Square is on the board, and it
        // contains one of the players' pieces.
   ct++;
   r += dirX;  // Go on to next square in this direction.
   c += dirY;
}

r = row - dirX;  // Now, look in the opposite direction.
c = col - dirY;
while ( r >= 0 && r < 13 && c >= 0 && c < 13
                                   && board[r][c] == player ) {
   ct++;
   r -= dirX;   // Go on to next square in this direction.
   c -= dirY;
}
```

Here is a picture of my program It uses a 13-by-13 board. You can do the same or use a normal 8-by-8 checkerboard.



Quiz on Chapter 7

1. What does the computer do when it executes the following statement? Try to give as complete an answer as possible.

```
Color[]  palette  =  new  Color[12];
```

2. What is meant by the *basetype* of an array?

3. What does it mean to sort an array?

4. What is the main advantage of binary search over linear search? What is the main disadvantage?

5. What is meant by a *dynamic array?* What is the advantage of a dynamic array over a regular array?

6. Suppose that a variable `strlst` has been declared as

```
ArrayList<String> strlst = new ArrayList<String>();
```

Assume that the list is not empty and that all the items in the list are non-null. Write a code segment that will find and print the string in the list that comes first in lexicographic order. How would your answer change if `strlst` were declared to be of type *ArrayList* instead of ArrayList<String>?

7. What is the purpose of the following subroutine? What is the meaning of the value that it returns, in terms of the value of its parameter?

```
static String concat( String[] str ) {
   if (str == null)
      return "";
   String ans = "";
   for (int i = 0; i < str.length; i++) {
      ans = ans + str[i];
   return ans;
}
```

8. Show the exact output produced by the following code segment.

```
char[][] pic = new char[6][6];
for (int i = 0; i < 6; i++)
   for (int j = 0; j < 6; j++) {
      if ( i == j  ||  i == 0  ||  i == 5 )
         pic[i][j] = '*';
      else
         pic[i][j] = '.';
   }
for (int i = 0; i < 6; i++) {
   for (int j = 0; j < 6; j++)
      System.out.print(pic[i][j]);
   System.out.println();
}
```

9. Write a complete static method that finds the largest value in an array of `ints`. The method should have one parameter, which is an array of type `int[]`. The largest number in the array should be returned as the value of the method.

10. Suppose that temperature measurements were made on each day of 1999 in each of 100 cities. The measurements have been stored in an array

```
int[][]  temps  =  new  int[100][365];
```

where `temps[c][d]` holds the measurement for city number `c` on the d^{th} day of the year. Write a code segment that will print out the average temperature, over the course of the whole year, for each city. The average temperature for a city can be obtained by adding up all 365 measurements for that city and dividing the answer by 365.0.

11. Suppose that a class, *Employee,* is defined as follows:

```
class Employee {
   String lastName;
   String firstName;
   double hourlyWage;
   int yearsWithCompany;
}
```

Suppose that data about 100 employees is **already** stored in an array:

```
Employee[] employeeData = new Employee[100];
```

Write a code segment that will output the first name, last name, and hourly wage of each employee who has been with the company for 20 years or more.

12. Suppose that `A` has been declared and initialized with the statement

```
double[] A = new double[20];
```

and suppose that `A` has **already** been filled with 20 values. Write a program segment that will find the average of all the **non-zero** numbers in the array. (The average is the sum of the numbers, divided by the number of numbers. Note that you will have to count the number of non-zero entries in the array.) Declare any variables that you use.

Chapter 8

Correctness, Robustness, Efficiency

In previous chapters, we have covered the fundamentals of programming. The chapters that follow this one will cover more advanced aspects of programming. The ideas that are presented will generally be more complex and the programs that use them a little more complicated. This relatively short chapter is a kind of turning point in which we look at the problem of getting such complex programs *right*.

Computer programs that fail are much too common. Programs are fragile. A tiny error can cause a program to misbehave or crash. Most of us are familiar with this from our own experience with computers. And we've all heard stories about software glitches that cause spacecraft to crash, telephone service to fail, and, in a few cases, people to die.

Programs don't have to be as bad as they are. It might well be impossible to guarantee that programs are problem-free, but careful programming and well-designed programming tools can help keep the problems to a minimum. This chapter will look at issues of correctness and robustness of programs. It also looks more closely at exceptions and the `try..catch` statement, and it introduces ***assertions***, another of the tools that Java provides as an aid in writing correct programs.

We will also look at another issue that is important for programs in the real world: efficiency. Even a completely correct program is not very useful if it takes an unreasonable amount of time to run. The last section of this chapter introduces techniques for analyzing the run time of algorithms.

8.1 Introduction to Correctness and Robustness

A program is ***correct*** if it accomplishes the task that it was designed to perform. It is ***robust*** if it can handle illegal inputs and other unexpected situations in a reasonable way. For example, consider a program that is designed to read some numbers from the user and then print the same numbers in sorted order. The program is correct if it works for any set of input numbers. It is robust if it can also deal with non-numeric input by, for example, printing an error message and ignoring the bad input. A non-robust program might crash or give nonsensical output in the same circumstance.

Every program should be correct. (A sorting program that doesn't sort correctly is pretty useless.) It's not the case that every program needs to be completely robust. It depends on who will use it and how it will be used. For example, a small utility program that you write for your own use doesn't have to be particularly robust.

The question of correctness is actually more subtle than it might appear. A programmer

works from a specification of what the program is supposed to do. The programmer's work is correct if the program meets its specification. But does that mean that the program itself is correct? What if the specification is incorrect or incomplete? A correct program should be a correct implementation of a complete and correct specification. The question is whether the specification correctly expresses the intention and desires of the people for whom the program is being written. This is a question that lies largely outside the domain of computer science.

8.1.1 Horror Stories

Most computer users have personal experience with programs that don't work or that crash. In many cases, such problems are just annoyances, but even on a personal computer there can be more serious consequences, such as lost work or lost money. When computers are given more important tasks, the consequences of failure can be proportionately more serious.

Just about a decade ago, the failure of two multi-million dollar space missions to Mars was prominent in the news. Both failures were probably due to software problems, but in both cases the problem was not with an incorrect program as such. In September 1999, the Mars Climate Orbiter burned up in the Martian atmosphere because data that was expressed in English units of measurement (such as feet and pounds) was entered into a computer program that was designed to use metric units (such as centimeters and grams). A few months later, the Mars Polar Lander probably crashed because its software turned off its landing engines too soon. The program was supposed to detect the bump when the spacecraft landed and turn off the engines then. It has been determined that deployment of the landing gear might have jarred the spacecraft enough to activate the program, causing it to turn off the engines when the spacecraft was still in the air. The unpowered spacecraft would then have fallen to the Martian surface. A more robust system would have checked the altitude before turning off the engines!

There are many equally dramatic stories of problems caused by incorrect or poorly written software. Let's look at a few incidents recounted in the book *Computer Ethics* by Tom Forester and Perry Morrison. (This book covers various ethical issues in computing. It, or something like it, is essential reading for any student of computer science.)

- In 1985 and 1986, one person was killed and several were injured by excess radiation, while undergoing radiation treatments by a mis-programmed computerized radiation machine. In another case, over a ten-year period ending in 1992, almost 1,000 cancer patients received radiation dosages that were 30% less than prescribed because of a programming error.
- In 1985, a computer at the Bank of New York started destroying records of on-going security transactions because of an error in a program. It took less than 24 hours to fix the program, but by that time, the bank was out $5,000,000 in overnight interest payments on funds that it had to borrow to cover the problem.
- The programming of the inertial guidance system of the F-16 fighter plane would have turned the plane upside-down when it crossed the equator, if the problem had not been discovered in simulation. The Mariner 18 space probe was lost because of an error in one line of a program. The Gemini V space capsule missed its scheduled landing target by a hundred miles, because a programmer forgot to take into account the rotation of the Earth.
- In 1990, AT&T's long-distance telephone service was disrupted throughout the United States when a newly loaded computer program proved to contain a bug.

Of course, there have been more recent problems. For example, computer software error contributed to the Northeast Blackout of 2003, one of the largest power outages in history. In 2006, the Airbus A380 was delayed by software incompatibility problems, at a cost of perhaps billions of dollars. In 2007, a software problem grounded thousands of planes at the Los Angeles International Airport. On May 6, 2010, a flaw in an automatic trading program apparently resulted in a 1000-point drop in the Dow Jones Industrial Average.

These are just a few examples. Software problems are all too common. As programmers, we need to understand why that is true and what can be done about it.

8.1.2 Java to the Rescue

Part of the problem, according to the inventors of Java, can be traced to programming languages themselves. Java was designed to provide some protection against certain types of errors. How can a language feature help prevent errors? Let's look at a few examples.

Early programming languages did not require variables to be declared. In such languages, when a variable name is used in a program, the variable is created automatically. You might consider this more convenient than having to declare every variable explicitly, but there is an unfortunate consequence: An inadvertent spelling error might introduce an extra variable that you had no intention of creating. This type of error was responsible, according to one famous story, for yet another lost spacecraft. In the `FORTRAN` programming language, the command "`DO 20 I = 1,5`" is the first statement of a counting loop. Now, spaces are insignificant in `FORTRAN`, so this is equivalent to "`DO20I=1,5`". On the other hand, the command "`DO20I=1.5`", with a period instead of a comma, is an assignment statement that assigns the value `1.5` to the variable `DO20I`. Supposedly, the inadvertent substitution of a period for a comma in a statement of this type caused a rocket to blow up on take-off. Because `FORTRAN` doesn't require variables to be declared, the compiler would be happy to accept the statement "`DO20I=1.5`." It would just create a new variable named `DO20I`. If `FORTRAN` required variables to be declared, the compiler would have complained that the variable `DO20I` was undeclared.

While most programming languages today do require variables to be declared, there are other features in common programming languages that can cause problems. Java has eliminated some of these features. Some people complain that this makes Java less efficient and less powerful. While there is some justice in this criticism, the increase in security and robustness is probably worth the cost in most circumstances. The best defense against some types of errors is to design a programming language in which the errors are impossible. In other cases, where the error can't be completely eliminated, the language can be designed so that when the error does occur, it will automatically be detected. This will at least prevent the error from causing further harm, and it will alert the programmer that there is a bug that needs fixing. Let's look at a few cases where the designers of Java have taken these approaches.

An array is created with a certain number of locations, numbered from zero up to some specified maximum index. It is an error to try to use an array location that is outside of the specified range. In Java, any attempt to do so is detected automatically by the system. In some other languages, such as C and C++, it's up to the programmer to make sure that the index is within the legal range. Suppose that an array, `A`, has three locations, `A[0]`, `A[1]`, and `A[2]`. Then `A[3]`, `A[4]`, and so on refer to memory locations beyond the end of the array. In Java, an attempt to store data in `A[3]` will be detected. The program will be terminated (unless the error is "caught", as discussed in Section 3.7). In C or C++, the computer will just go ahead and store the data in memory that is not part of the array. Since there is no telling what that memory location is being used for, the result will be unpredictable. The consequences could

be much more serious than a terminated program. (See, for example, the discussion of buffer overflow errors later in this section.)

Pointers are a notorious source of programming errors. In Java, a variable of object type holds either a pointer to an object or the special value `null`. Any attempt to use a `null` value as if it were a pointer to an actual object will be detected by the system. In some other languages, again, it's up to the programmer to avoid such null pointer errors. In my old Macintosh computer, a `null` pointer was actually implemented as if it were a pointer to memory location zero. A program could use a null pointer to change values stored in memory near location zero. Unfortunately, the Macintosh stored important system data in those locations. Changing that data could cause the whole system to crash, a consequence more severe than a single failed program.

Another type of pointer error occurs when a pointer value is pointing to an object of the wrong type or to a segment of memory that does not even hold a valid object at all. These types of errors are impossible in Java, which does not allow programmers to manipulate pointers directly. In other languages, it is possible to set a pointer to point, essentially, to any location in memory. If this is done incorrectly, then using the pointer can have unpredictable results.

Another type of error that cannot occur in Java is a memory leak. In Java, once there are no longer any pointers that refer to an object, that object is "garbage collected" so that the memory that it occupied can be reused. In other languages, it is the programmer's responsibility to return unused memory to the system. If the programmer fails to do this, unused memory can build up, leaving less memory for programs and data. There is a story that many common programs for older Windows computers had so many memory leaks that the computer would run out of memory after a few days of use and would have to be restarted.

Many programs have been found to suffer from ***buffer overflow errors***. Buffer overflow errors often make the news because they are responsible for many network security problems. When one computer receives data from another computer over a network, that data is stored in a buffer. The buffer is just a segment of memory that has been allocated by a program to hold data that it expects to receive. A buffer overflow occurs when more data is received than will fit in the buffer. The question is, what happens then? If the error is detected by the program or by the networking software, then the only thing that has happened is a failed network data transmission. The real problem occurs when the software does not properly detect buffer overflows. In that case, the software continues to store data in memory even after the buffer is filled, and the extra data goes into some part of memory that was not allocated by the program as part of the buffer. That memory might be in use for some other purpose. It might contain important data. It might even contain part of the program itself. This is where the real security issues come in. Suppose that a buffer overflow causes part of a program to be replaced with extra data received over a network. When the computer goes to execute the part of the program that was replaced, it's actually executing data that was received from another computer. That data could be anything. It could be a program that crashes the computer or takes it over. A malicious programmer who finds a convenient buffer overflow error in networking software can try to exploit that error to trick other computers into executing his programs.

For software written completely in Java, buffer overflow errors are impossible. The language simply does not provide any way to store data into memory that has not been properly allocated. To do that, you would need a pointer that points to unallocated memory or you would have to refer to an array location that lies outside the range allocated for the array. As explained above, neither of these is possible in Java. (However, there could conceivably still be errors in Java's standard classes, since some of the methods in these classes are actually written in the

C programming language rather than in Java.)

It's clear that language design can help prevent errors or detect them when they occur. Doing so involves restricting what a programmer is allowed to do. Or it requires tests, such as checking whether a pointer is `null`, that take some extra processing time. Some programmers feel that the sacrifice of power and efficiency is too high a price to pay for the extra security. In some applications, this is true. However, there are many situations where safety and security are primary considerations. Java is designed for such situations.

8.1.3 Problems Remain in Java

There is one area where the designers of Java chose not to detect errors automatically: numerical computations. In Java, a value of type **int** is represented as a 32-bit binary number. With 32 bits, it's possible to represent a little over four billion different values. The values of type **int** range from -2147483648 to 2147483647. What happens when the result of a computation lies outside this range? For example, what is `2147483647 + 1`? And what is `2000000000 * 2`? The mathematically correct result in each case cannot be represented as a value of type **int**. These are examples of ***integer overflow***. In most cases, integer overflow should be considered an error. However, Java does not automatically detect such errors. For example, it will compute the value of `2147483647 + 1` to be the negative number, `-2147483648`. (What happens is that any extra bits beyond the 32-nd bit in the correct answer are discarded. Values greater than 2147483647 will "wrap around" to negative values. Mathematically speaking, the result is always "correct modulo 2^{32}.")

For example, consider the 3N+1 program, which was discussed in Subsection 3.2.2. Starting from a positive integer `N`, the program computes a certain sequence of integers:

```
while ( N != 1 ) {
   if ( N % 2 == 0 )  // If N is even...
      N = N / 2;
   else
      N = 3 * N + 1;
   System.out.println(N);
}
```

But there is a problem here: If `N` is too large, then the value of `3*N+1` will not be mathematically correct because of integer overflow. The problem arises whenever `3*N+1 > 2147483647`, that is when `N > 2147483646/3`. For a completely correct program, we should check for this possibility **before** computing `3*N+1`:

```
while ( N != 1 ) {
   if ( N % 2 == 0 )  // If N is even...
      N = N / 2;
   else {
      if (N > 2147483646/3) {
         System.out.println("Sorry, but the value of N has become");
         System.out.println("too large for your computer!");
         break;
      }
      N = 3 * N + 1;
   }
   System.out.println(N);
}
```

The problem here is not that the original algorithm for computing `3N+1` sequences was wrong. The problem is that it just can't be correctly implemented using 32-bit integers. Many programs ignore this type of problem. But integer overflow errors have been responsible for their share of serious computer failures, and a completely robust program should take the possibility of integer overflow into account. (The infamous "Y2K" bug was, in fact, just this sort of error.)

For numbers of type **double**, there are even more problems. There are still overflow errors, which occur when the result of a computation is outside the range of values that can be represented as a value of type **double**. This range extends up to about 1.7 times 10 to the power 308. Numbers beyond this range do not "wrap around" to negative values. Instead, they are represented by special values that have no real numerical equivalent. The special values `Double.POSITIVE_INFINITY` and `Double.NEGATIVE_INFINITY` represent numbers outside the range of legal values. For example, `20 * 1e308` is computed to be `Double.POSITIVE_INFINITY`. Another special value of type **double**, `Double.NaN`, represents an illegal or undefined result. ("NaN" stands for "Not a Number".) For example, the result of dividing zero by zero or taking the square root of a negative number is `Double.NaN`. You can test whether a number `x` is this special not-a-number value by calling the **boolean**-valued function `Double.isNaN(x)`.

For real numbers, there is the added complication that most real numbers can only be represented approximately on a computer. A real number can have an infinite number of digits after the decimal point. A value of type **double** is only accurate to about 15 digits. The real number 1/3, for example, is the repeating decimal `0.333333333333...`, and there is no way to represent it exactly using a finite number of digits. Computations with real numbers generally involve a loss of accuracy. In fact, if care is not exercised, the result of a large number of such computations might be completely wrong! There is a whole field of computer science, known as ***numerical analysis***, which is devoted to studying algorithms that manipulate real numbers.

So you see that not all possible errors are avoided or detected automatically in Java. Furthermore, even when an error is detected automatically, the system's default response is to report the error and terminate the program. This is hardly robust behavior! So, a Java programmer still needs to learn techniques for avoiding and dealing with errors. These are the main topics of the next three sections.

8.2 Writing Correct Programs

Correct programs don't just happen. It takes planning and attention to detail to avoid errors in programs. There are some techniques that programmers can use to increase the likelihood that their programs are correct.

8.2.1 Provably Correct Programs

In some cases, it is possible to **prove** that a program is correct. That is, it is possible to demonstrate mathematically that the sequence of computations represented by the program will always produce the correct result. Rigorous proof is difficult enough that in practice it can only be applied to fairly small programs. Furthermore, it depends on the fact that the "correct result" has been specified correctly and completely. As I've already pointed out, a program that correctly meets its specification is not useful if its specification was wrong. Nevertheless, even in everyday programming, we can apply some of the ideas and techniques that are used in proving that programs are correct.

The fundamental ideas are ***process*** and ***state***. A state consists of all the information

relevant to the execution of a program at a given moment during its execution. The state includes, for example, the values of all the variables in the program, the output that has been produced, any input that is waiting to be read, and a record of the position in the program where the computer is working. A process is the sequence of states that the computer goes through as it executes the program. From this point of view, the meaning of a statement in a program can be expressed in terms of the effect that the execution of that statement has on the computer's state. As a simple example, the meaning of the assignment statement "`x = 7;`" is that after this statement is executed, the value of the variable `x` will be `7`. We can be absolutely sure of this fact, so it is something upon which we can build part of a mathematical proof.

In fact, it is often possible to look at a program and deduce that some fact must be true at a given point during the execution of a program. For example, consider the `do` loop:

```
do {
   TextIO.put("Enter a positive integer: ");
   N = TextIO.getlnInt();
} while (N <= 0);
```

After this loop ends, we can be absolutely sure that the value of the variable `N` is greater than zero. The loop cannot end until this condition is satisfied. This fact is part of the meaning of the `while` loop. More generally, if a `while` loop uses the test "`while (`⟨*condition*⟩`)`", then after the loop ends, we can be sure that the ⟨*condition*⟩ is false. We can then use this fact to draw further deductions about what happens as the execution of the program continues. (With a loop, by the way, we also have to worry about the question of whether the loop will ever end. This is something that has to be verified separately.)

A fact that can be proven to be true after a given program segment has been executed is called a ***postcondition*** of that program segment. Postconditions are known facts upon which we can build further deductions about the behavior of the program. A postcondition of a program as a whole is simply a fact that can be proven to be true after the program has finished executing. A program can be proven to be correct by showing that the postconditions of the program meet the program's specification.

Consider the following program segment, where all the variables are of type **double**:

```
disc = B*B - 4*A*C;
x = (-B + Math.sqrt(disc)) / (2*A);
```

The quadratic formula (from high-school mathematics) assures us that the value assigned to `x` is a solution of the equation `A*x`^{2} `+ B*x + C = 0`, provided that the value of `disc` is greater than or equal to zero and the value of `A` is not zero. **If** we can assume or guarantee that `B*B-4*A*C >= 0` and that `A != 0`, then the fact that `x` is a solution of the equation becomes a postcondition of the program segment. We say that the condition, `B*B-4*A*C >= 0` is a ***precondition*** of the program segment. The condition that `A != 0` is another precondition. A precondition is defined to be a condition that must be true at a given point in the execution of a program in order for the program to continue correctly. A precondition is something that you want to be true. It's something that you have to check or force to be true, if you want your program to be correct.

We've encountered preconditions and postconditions once before, in Subsection 4.6.1. That section introduced preconditions and postconditions as a way of specifying the contract of a subroutine. As the terms are being used here, a precondition of a subroutine is just a precondition of the code that makes up the definition of the subroutine, and the postcondition of a subroutine is a postcondition of the same code. In this section, we have generalized these terms to make them more useful in talking about program correctness.

Let's see how this works by considering a longer program segment:

```
do {
   TextIO.putln("Enter A, B, and C.  B*B-4*A*C must be >= 0.");
   TextIO.put("A = ");
   A = TextIO.getlnDouble();
   TextIO.put("B = ");
   B = TextIO.getlnDouble();
   TextIO.put("C = ");
   C = TextIO.getlnDouble();
   if (A == 0 || B*B - 4*A*C < 0)
      TextIO.putln("Your input is illegal.  Try again.");
} while (A == 0 || B*B - 4*A*C < 0);

disc = B*B - 4*A*C;
x = (-B + Math.sqrt(disc)) / (2*A);
```

After the loop ends, we can be sure that `B*B-4*A*C >= 0` and that `A != 0`. The preconditions for the last two lines are fulfilled, so the postcondition that `x` is a solution of the equation $\texttt{A*x}^2$ `+ B*x + C = 0` is also valid. This program segment correctly and provably computes a solution to the equation. (Actually, because of problems with representing numbers on computers, this is not 100% true. The **algorithm** is correct, but the **program** is not a perfect implementation of the algorithm. See the discussion in Subsection 8.1.3.)

Here is another variation, in which the precondition is checked by an `if` statement. In the first part of the `if` statement, where a solution is computed and printed, we know that the preconditions are fulfilled. In the other parts, we know that one of the preconditions fails to hold. In any case, the program is correct.

```
TextIO.putln("Enter your values for A, B, and C.");
TextIO.put("A = ");
A = TextIO.getlnDouble();
TextIO.put("B = ");
B = TextIO.getlnDouble();
TextIO.put("C = ");
C = TextIO.getlnDouble();

if (A != 0 && B*B - 4*A*C >= 0) {
   disc = B*B - 4*A*C;
   x = (-B + Math.sqrt(disc)) / (2*A);
   TextIO.putln("A solution of A*X*X + B*X + C = 0 is " + x);
}
else if (A == 0) {
   TextIO.putln("The value of A cannot be zero.");
}
else {
   TextIO.putln("Since B*B - 4*A*C is less than zero, the");
   TextIO.putln("equation A*X*X + B*X + C = 0 has no solution.");
}
```

Whenever you write a program, it's a good idea to watch out for preconditions and think about how your program handles them. Often, a precondition can offer a clue about how to write the program.

For example, every array reference, such as `A[i]`, has a precondition: The index must be within the range of legal indices for the array. For `A[i]`, the precondition is that `0 <= i`

`< A.length`. The computer will check this condition when it evaluates `A[i]`, and if the condition is not satisfied, the program will be terminated. In order to avoid this, you need to make sure that the index has a legal value. (There is actually another precondition, namely that `A` is not `null`, but let's leave that aside for the moment.) Consider the following code, which searches for the number `x` in the array `A` and sets the value of `i` to be the index of the array element that contains `x`:

```
i = 0;
while (A[i] != x) {
   i++;
}
```

As this program segment stands, it has a precondition, namely that `x` is actually in the array. If this precondition is satisfied, then the loop will end when `A[i] == x`. That is, the value of `i` when the loop ends will be the position of `x` in the array. However, if `x` is not in the array, then the value of `i` will just keep increasing until it is equal to `A.length`. At that time, the reference to `A[i]` is illegal and the program will be terminated. To avoid this, we can add a test to make sure that the precondition for referring to `A[i]` is satisfied:

```
i = 0;
while (i < A.length && A[i] != x) {
   i++;
}
```

Now, the loop will definitely end. After it ends, `i` will satisfy **either** `i == A.length` or `A[i] == x`. An `if` statement can be used after the loop to test which of these conditions caused the loop to end:

```
i = 0;
while (i < A.length && A[i] != x) {
   i++;
}

if (i == A.length)
   System.out.println("x is not in the array");
else
   System.out.println("x is in position " + i);
```

8.2.2 Robust Handling of Input

One place where correctness and robustness are important—and especially difficult—is in the processing of input data, whether that data is typed in by the user, read from a file, or received over a network. Files and networking will be covered in Chapter 11, which will make essential use of material that will be covered in the next section of this chapter. For now, let's look at an example of processing user input.

Examples in this textbook use my *TextIO* class for reading input from the user. This class has built-in error handling. For example, the function `TextIO.getDouble()` is guaranteed to return a legal value of type **double**. If the user types an illegal value, then *TextIO* will ask the user to re-enter their response; your program never sees the illegal value. However, this approach can be clumsy and unsatisfactory, especially when the user is entering complex data. In the following example, I'll do my own error-checking.

Sometimes, it's useful to be able to look ahead at what's coming up in the input without actually reading it. For example, a program might need to know whether the next item in

the input is a number or a word. For this purpose, the *TextIO* class includes the function `TextIO.peek()`. This function returns a **char** which is the next character in the user's input, but it does not actually read that character. If the next thing in the input is an end-of-line, then `TextIO.peek()` returns the new-line character, `'\n'`.

Often, what we really need to know is the next **non-blank** character in the user's input. Before we can test this, we need to skip past any spaces (and tabs). Here is a function that does this. It uses `TextIO.peek()` to look ahead, and it reads characters until the next character in the input is either an end-of-line or some non-blank character. (The function `TextIO.getAnyChar()` reads and returns the next character in the user's input, even if that character is a space. By contrast, the more common `TextIO.getChar()` would skip any blanks and then read and return the next non-blank character. We can't use `TextIO.getChar()` here since the object is to skip the blanks **without** reading the next non-blank character.)

```
/**
 * Reads past any blanks and tabs in the input.
 * Postcondition:  The next character in the input is an
 *                 end-of-line or a non-blank character.
 */
static void skipBlanks() {
   char ch;
   ch = TextIO.peek();
   while (ch == ' ' || ch == '\t') {
         // Next character is a space or tab; read it
         // and look at the character that follows it.
      ch = TextIO.getAnyChar();
      ch = TextIO.peek();
   }
} // end skipBlanks()
```

(In fact, this operation is so common that it is built into the most recent version of `TextIO`. The method `TextIO.skipBlanks()` does essentially the same thing as the `skipBlanks()` method presented here.)

An example in Subsection 3.5.3 allowed the user to enter length measurements such as "3 miles" or "1 foot". It would then convert the measurement into inches, feet, yards, and miles. But people commonly use combined measurements such as "3 feet 7 inches". Let's improve the program so that it allows inputs of this form.

More specifically, the user will input lines containing one or more measurements such as "1 foot" or "3 miles 20 yards 2 feet". The legal units of measure are inch, foot, yard, and mile. The program will also recognize plurals (inches, feet, yards, miles) and abbreviations (in, ft, yd, mi). Let's write a subroutine that will read one line of input of this form and compute the equivalent number of inches. The main program uses the number of inches to compute the equivalent number of feet, yards, and miles. If there is any error in the input, the subroutine will print an error message and return the value `-1`. The subroutine assumes that the input line is not empty. The main program tests for this before calling the subroutine and uses an empty line as a signal for ending the program.

Ignoring the possibility of illegal inputs, a pseudocode algorithm for the subroutine is

```
inches = 0    // This will be the total number of inches
while there is more input on the line:
    read the numerical measurement
    read the units of measure
```

```
        add the measurement to inches
    return inches
```

We can test whether there is more input on the line by checking whether the next non-blank character is the end-of-line character. But this test has a precondition: Before we can test the next non-blank character, we have to skip over any blanks. So, the algorithm becomes

```
inches = 0
skipBlanks()
while TextIO.peek() is not '\n':
    read the numerical measurement
    read the unit of measure
    add the measurement to inches
    skipBlanks()
return inches
```

Note the call to `skipBlanks()` at the end of the `while` loop. This subroutine must be executed before the computer returns to the test at the beginning of the loop. More generally, if the test in a `while` loop has a precondition, then you have to make sure that this precondition holds at the **end** of the while loop, before the computer jumps back to re-evaluate the test, as well as before the start of the loop.

What about error checking? Before reading the numerical measurement, we have to make sure that there is really a number there to read. Before reading the unit of measure, we have to test that there is something there to read. (The number might have been the last thing on the line. An input such as "3", without a unit of measure, is not acceptable.) Also, we have to check that the unit of measure is one of the valid units: inches, feet, yards, or miles. Here is an algorithm that includes error-checking:

```
inches = 0
skipBlanks()

while TextIO.peek() is not '\n':

    if the next character is not a digit:
       report an error and return -1
    Let measurement = TextIO.getDouble();

    skipBlanks()    // Precondition for the next test!!
    if the next character is end-of-line:
       report an error and return -1
    Let units = TextIO.getWord()

    if the units are inches:
        add measurement to inches
    else if the units are feet:
        add 12*measurement to inches
    else if the units are yards:
        add 36*measurement to inches
    else if the units are miles:
        add 12*5280*measurement to inches
    else
        report an error and return -1

    skipBlanks()

return inches
```

As you can see, error-testing adds significantly to the complexity of the algorithm. Yet this is still a fairly simple example, and it doesn't even handle all the possible errors. For example, if the user enters a numerical measurement such as `1e400` that is outside the legal range of values of type **double**, then the program will fall back on the default error-handling in *TextIO*. Something even more interesting happens if the measurement is "1e308 miles". The number `1e308` is legal, but the corresponding number of inches is outside the legal range of values for type **double**. As mentioned in the previous section, the computer will get the value `Double.POSITIVE_INFINITY` when it does the computation.

Here is the subroutine written out in Java:

```
/**
 * Reads the user's input measurement from one line of input.
 * Precondition:   The input line is not empty.
 * Postcondition:  If the user's input is legal, the measurement
 *                 is converted to inches and returned.  If the
 *                 input is not legal, the value -1 is returned.
 *                 The end-of-line is NOT read by this routine.
 */
static double readMeasurement() {

   double inches;  // Total number of inches in user's measurement.

   double measurement;  // One measurement,
                        //   such as the 12 in "12 miles"
   String units;        // The units specified for the measurement,
                        //   such as "miles"

   char ch;  // Used to peek at next character in the user's input.

   inches = 0;  // No inches have yet been read.

   skipBlanks();
   ch = TextIO.peek();

   /* As long as there is more input on the line, read a measurement and
      add the equivalent number of inches to the variable, inches.  If an
      error is detected during the loop, end the subroutine immediately
      by returning -1. */

   while (ch != '\n') {

       /* Get the next measurement and the units.  Before reading
          anything, make sure that a legal value is there to read. */

       if ( ! Character.isDigit(ch) ) {
           TextIO.putln(
                 "Error:  Expected to find a number, but found " + ch);
           return -1;
       }
       measurement = TextIO.getDouble();

       skipBlanks();
       if (TextIO.peek() == '\n') {
           TextIO.putln(
                 "Error:  Missing unit of measure at end of line.");
           return -1;
       }
```

```
            units = TextIO.getWord();
            units = units.toLowerCase();

            /* Convert the measurement to inches and add it to the total. */

            if (units.equals("inch")
                    || units.equals("inches") || units.equals("in")) {
                inches += measurement;
            }
            else if (units.equals("foot")
                       || units.equals("feet") || units.equals("ft")) {
                inches += measurement * 12;
            }
            else if (units.equals("yard")
                       || units.equals("yards") || units.equals("yd")) {
                inches += measurement * 36;
            }
            else if (units.equals("mile")
                       || units.equals("miles") || units.equals("mi")) {
                inches += measurement * 12 * 5280;
            }
            else {
                TextIO.putln("Error: \"" + units
                                  + "\" is not a legal unit of measure.");
                return -1;
            }

            /* Look ahead to see whether the next thing on the line is
               the end-of-line. */

            skipBlanks();
            ch = TextIO.peek();

        }  // end while

        return inches;

    } // end readMeasurement()
```

The source code for the complete program can be found in the file *LengthConverter2.java*.

8.3 Exceptions and try..catch

Getting a program to work under ideal circumstances is usually a lot easier than making the program ***robust***. A robust program can survive unusual or "exceptional" circumstances without crashing. One approach to writing robust programs is to anticipate the problems that might arise and to include tests in the program for each possible problem. For example, a program will crash if it tries to use an array element `A[i]`, when `i` is not within the declared range of indices for the array `A`. A robust program must anticipate the possibility of a bad index and guard against it. One way to do this is to write the program in a way that ensures (as a postcondition of the code that precedes the array reference) that the index is in the legal range. Another way is to test whether the index value is legal before using it in the array. This could be done with an `if` statement:

```
if (i < 0 || i >= A.length) {
   ...  // Do something to handle the out-of-range index, i
}
else {
   ...  // Process the array element, A[i]
}
```

There are some problems with this approach. It is difficult and sometimes impossible to anticipate all the possible things that might go wrong. It's not always clear what to do when an error is detected. Furthermore, trying to anticipate all the possible problems can turn what would otherwise be a straightforward program into a messy tangle of `if` statements.

8.3.1 Exceptions and Exception Classes

We have already seen in Section 3.7 that Java (like its cousin, C++) provides a neater, more structured alternative technique for dealing with errors that can occur while a program is running. The technique is referred to as ***exception handling***. The word "exception" is meant to be more general than "error." It includes any circumstance that arises as the program is executed which is meant to be treated as an exception to the normal flow of control of the program. An exception might be an error, or it might just be a special case that you would rather not have clutter up your elegant algorithm.

When an exception occurs during the execution of a program, we say that the exception is ***thrown***. When this happens, the normal flow of the program is thrown off-track, and the program is in danger of crashing. However, the crash can be avoided if the exception is ***caught*** and handled in some way. An exception can be thrown in one part of a program and caught in a different part. An exception that is not caught will generally cause the program to crash. (More exactly, the thread that throws the exception will crash. In a multithreaded program, it is possible for other threads to continue even after one crashes. We will cover threads in Chapter 12. In particular, GUI programs are multithreaded, and parts of the program might continue to function even while other parts are non-functional because of exceptions.)

By the way, since Java programs are executed by a Java interpreter, having a program crash simply means that it terminates abnormally and prematurely. It doesn't mean that the Java interpreter will crash. In effect, the interpreter catches any exceptions that are not caught by the program. The interpreter responds by terminating the program. In many other programming languages, a crashed program will sometimes crash the entire system and freeze the computer until it is restarted. With Java, such system crashes should be impossible—which means that when they happen, you have the satisfaction of blaming the system rather than your own program.

Exceptions were introduced in Section 3.7, along with the `try..catch` statement, which is used to catch and handle exceptions. However, that section did not cover the complete syntax of `try..catch` or the full complexity of exceptions. In this section, we cover these topics in full detail.

* * *

When an exception occurs, the thing that is actually "thrown" is an object. This object can carry information (in its instance variables) from the point where the exception occurs to the point where it is caught and handled. This information always includes the ***subroutine call stack***, which is a list of the subroutines that were being executed when the exception was thrown. (Since one subroutine can call another, several subroutines can be active at the same time.) Typically, an exception object also includes an error message describing what happened

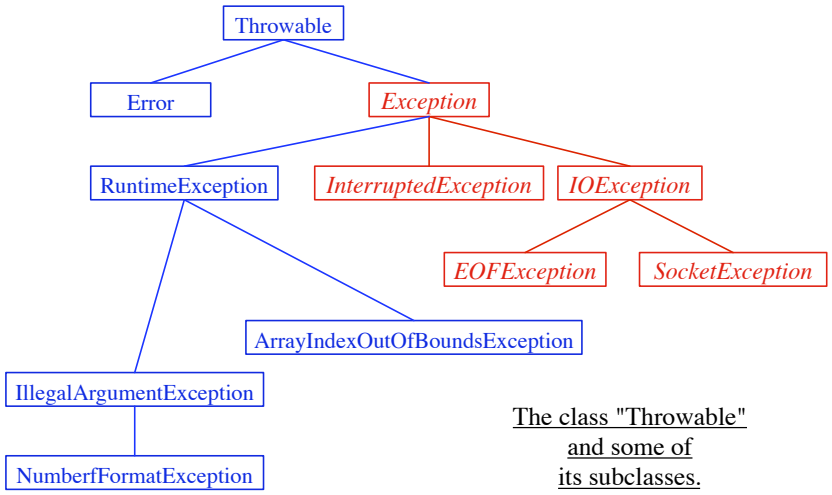
to cause the exception, and it can contain other data as well. All exception objects must belong to a subclass of the standard class `java.lang.Throwable`. In general, each different type of exception is represented by its own subclass of *Throwable*, and these subclasses are arranged in a fairly complex class hierarchy that shows the relationship among various types of exception. *Throwable* has two direct subclasses, *Error* and *Exception*. These two subclasses in turn have many other predefined subclasses. In addition, a programmer can create new exception classes to represent new types of exception.

Most of the subclasses of the class *Error* represent serious errors within the Java virtual machine that should ordinarily cause program termination because there is no reasonable way to handle them. In general, you should not try to catch and handle such errors. An example is a *ClassFormatError*, which occurs when the Java virtual machine finds some kind of illegal data in a file that is supposed to contain a compiled Java class. If that class was being loaded as part of the program, then there is really no way for the program to proceed.

On the other hand, subclasses of the class *Exception* represent exceptions that are meant to be caught. In many cases, these are exceptions that might naturally be called "errors," but they are errors in the program or in input data that a programmer can anticipate and possibly respond to in some reasonable way. (However, you should avoid the temptation of saying, "Well, I'll just put a thing here to catch all the errors that might occur, so my program won't crash." If you don't have a reasonable way to respond to the error, it's best just to let the program crash, because trying to go on will probably only lead to worse things down the road—in the worst case, a program that gives an incorrect answer without giving you any indication that the answer might be wrong!)

The class *Exception* has its own subclass, *RuntimeException*. This class groups together many common exceptions, including all those that have been covered in previous sections. For example, *IllegalArgumentException* and *NullPointerException* are subclasses of *RuntimeException*. A *RuntimeException* generally indicates a bug in the program, which the programmer should fix. *RuntimeExceptions* and *Errors* share the property that a program can simply ignore the possibility that they might occur. ("Ignoring" here means that you are content to let your program crash if the exception occurs.) For example, a program does this every time it uses an array reference like `A[i]` without making arrangements to catch a possible *ArrayIndexOutOfBoundsException*. For all other exception classes besides *Error*, *RuntimeException*, and their subclasses, exception-handling is "mandatory" in a sense that I'll discuss below.

The following diagram is a class hierarchy showing the class `Throwable` and just a few of its subclasses. Classes that require mandatory exception-handling are shown in italic:



The class *Throwable* includes several instance methods that can be used with any exception object. If `e` is of type *Throwable* (or one of its subclasses), then `e.getMessage()` is a function that returns a *String* that describes the exception. The function `e.toString()`, which is used by the system whenever it needs a string representation of the object, returns a *String* that contains the name of the class to which the exception belongs as well as the same string that would be returned by `e.getMessage()`. And the method `e.printStackTrace()` writes a stack trace to standard output that tells which subroutines were active when the exception occurred. A stack trace can be very useful when you are trying to determine the cause of the problem. (Note that if an exception is **not** caught by the program, then the default response to the exception prints the stack trace to standard output.)

8.3.2 The try Statement

To catch exceptions in a Java program, you need a `try` statement. We have been using such statements since Section 3.7, but the full syntax of the `try` statement is more complicated than what was presented there. The `try` statements that we have used so far had a syntax similar to the following example:

```
try {
    double determinant = M[0][0]*M[1][1] - M[0][1]*M[1][0];
    System.out.println("The determinant of M is " + determinant);
}
catch ( ArrayIndexOutOfBoundsException e ) {
   System.out.println("M is the wrong size to have a determinant.");
   e.printStackTrace();
}
```

Here, the computer tries to execute the block of statements following the word "`try`". If no exception occurs during the execution of this block, then the "`catch`" part of the statement is simply ignored. However, if an exception of type *ArrayIndexOutOfBoundsException* occurs, then the computer jumps immediately to the `catch` clause of the `try` statement. This block of statements is said to be an ***exception handler*** for *ArrayIndexOutOfBoundsException*. By handling the exception in this way, you prevent it from crashing the program. Before the body of the `catch` clause is executed, the object that represents the exception is assigned to the variable `e`, which is used in this example to print a stack trace.

However, the full syntax of the `try` statement allows more than one `catch` clause. This makes it possible to catch several different types of exception with one `try` statement. In the above example, in addition to the possible *ArrayIndexOutOfBoundsException*, there is a possible *NullPointerException* which will occur if the value of `M` is `null`. We can handle both possible exceptions by adding a second `catch` clause to the `try` statement:

```
try {
    double determinant = M[0][0]*M[1][1] - M[0][1]*M[1][0];
    System.out.println("The determinant of M is " + determinant);
}
catch ( ArrayIndexOutOfBoundsException e ) {
   System.out.println("M is the wrong size to have a determinant.");
}
catch ( NullPointerException e ) {
   System.out.print("Programming error!  M doesn't exist." + );
}
```

Here, the computer tries to execute the statements in the `try` clause. If no error occurs, both of the `catch` clauses are skipped. If an *ArrayIndexOutOfBoundsException* occurs, the computer executes the body of the first `catch` clause and skips the second one. If a *NullPointerException* occurs, it jumps to the second `catch` clause and executes that.

Note that both *ArrayIndexOutOfBoundsException* and *NullPointerException* are subclasses of *RuntimeException*. It's possible to catch all *RuntimeExceptions* with a single `catch` clause. For example:

```
try {
    double determinant = M[0][0]*M[1][1] - M[0][1]*M[1][0];
    System.out.println("The determinant of M is " + determinant);
}
catch ( RuntimeException err ) {
   System.out.println("Sorry, an error has occurred.");
   System.out.println("The error was: " + err);
}
```

The `catch` clause in this `try` statement will catch any exception belonging to class *RuntimeException* or to any of its subclasses. This shows why exception classes are organized into a class hierarchy. It allows you the option of casting your net narrowly to catch only a specific type of exception. Or you can cast your net widely to catch a wide class of exceptions. Because of subclassing, when there are multiple `catch` clauses in a `try` statement, it is possible that a given exception might match several of those `catch` clauses. For example, an exception of type *NullPointerException* would match `catch` clauses for *NullPointerException*, *RuntimeException*, *Exception*, or *Throwable*. In this case, only the **first** `catch` clause that matches the exception is executed.

The example I've given here is not particularly realistic. You are not very likely to use exception-handling to guard against null pointers and bad array indices. This is a case where careful programming is better than exception handling: Just be sure that your program assigns a reasonable, non-`null` value to the array `M`. You would certainly resent it if the designers of Java forced you to set up a `try..catch` statement every time you wanted to use an array! This is why handling of potential *RuntimeExceptions* is not mandatory. There are just too many things that might go wrong! (This also shows that exception-handling does not solve the problem of program robustness. It just gives you a tool that will in many cases let you approach the problem in a more organized way.)

* * *

I have still not completely specified the syntax of the `try` statement. There is one additional element: the possibility of a ***finally clause*** at the end of a `try` statement. The complete syntax of the `try` statement can be described as:

```
try {
   ⟨statements⟩
}
⟨optional-catch-clauses⟩
⟨optional-finally-clause⟩
```

Note that the `catch` clauses are also listed as optional. The `try` statement can include zero or more `catch` clauses and, optionally, a `finally` clause. The `try` statement **must** include one or the other. That is, a `try` statement can have either a `finally` clause, or one or more `catch` clauses, or both. The syntax for a `catch` clause is

```
catch ( ⟨exception-class-name⟩ ⟨variable-name⟩ ) {
   ⟨statements⟩
}
```

and the syntax for a `finally` clause is

```
finally {
   ⟨statements⟩
}
```

The semantics of the `finally` clause is that the block of statements in the `finally` clause is guaranteed to be executed as the last step in the execution of the try statement, whether or not any exception occurs and whether or not any exception that does occur is caught and handled. The `finally` clause is meant for doing essential cleanup that under no circumstances should be omitted. One example of this type of cleanup is closing a network connection. Although you don't yet know enough about networking to look at the actual programming in this case, we can consider some pseudocode:

```
try {
   open a network connection
}
catch ( IOException e ) {
   report the error
   return  // Don't continue if connection can't be opened!
}

// At this point, we KNOW that the connection is open.

try {
   communicate over the connection
}
catch ( IOException e ) {
   handle the error
}
finally {
   close the connection
}
```

The `finally` clause in the second `try` statement ensures that the network connection will definitely be closed, whether or not an error occurs during the communication. The first `try` statement is there to make sure that we don't even try to communicate over the network unless we have successfully opened a connection. The pseudocode in this example follows a general pattern that can be used to robustly obtain a resource, use the resource, and then release the resource.

8.3.3 Throwing Exceptions

There are times when it makes sense for a program to deliberately throw an exception. This is the case when the program discovers some sort of exceptional or error condition, but there is no reasonable way to handle the error at the point where the problem is discovered. The program can throw an exception in the hope that some other part of the program will catch and handle the exception. This can be done with a ***throw statement***. You have already seen an example of this in Subsection 4.3.5. In this section, we cover the `throw` statement more fully. The syntax of the `throw` statement is:

```
throw ⟨exception-object⟩ ;
```

The ⟨*exception-object*⟩ must be an object belonging to one of the subclasses of `Throwable`. Usually, it will in fact belong to one of the subclasses of `Exception`. In most cases, it will be a newly constructed object created with the `new` operator. For example:

```
throw new ArithmeticException("Division by zero");
```

The parameter in the constructor becomes the error message in the exception object; if `e` refers to the object, the error message can be retrieved by calling `e.getMessage()`. (You might find this example a bit odd, because you might expect the system itself to throw an *ArithmeticException* when an attempt is made to divide by zero. So why should a programmer bother to throw the exception? Recall that if the numbers that are being divided are of type **int**, then division by zero will indeed throw an *ArithmeticException*. However, no arithmetic operations with floating-point numbers will ever produce an exception. Instead, the special value `Double.NaN` is used to represent the result of an illegal operation. In some situations, you might prefer to throw an *ArithmeticException* when a real number is divided by zero.)

An exception can be thrown either by the system or by a `throw` statement. The exception is processed in exactly the same way in either case. Suppose that the exception is thrown inside a `try` statement. If that `try` statement has a `catch` clause that handles that type of exception, then the computer jumps to the `catch` clause and executes it. The exception has been ***handled***. After handling the exception, the computer executes the `finally` clause of the `try` statement, if there is one. It then continues normally with the rest of the program, which follows the `try` statement. If the exception is not immediately caught and handled, the processing of the exception will continue.

When an exception is thrown during the execution of a subroutine and the exception is not handled in the same subroutine, then that subroutine is terminated (after the execution of any pending `finally` clauses). Then the routine that called that subroutine gets a chance to handle the exception. That is, if the subroutine was called inside a `try` statement that has an appropriate `catch` clause, then **that** `catch` clause will be executed and the program will continue on normally from there. Again, if the second routine does not handle the exception, then it also is terminated and the routine that called **it** (if any) gets the next shot at the exception. The exception will crash the program only if it passes up through the entire chain of

subroutine calls without being handled. (In fact, even this is not quite true: In a multithreaded program, only the thread in which the exception occurred is terminated.)

A subroutine that might generate an exception can announce this fact by adding a clause "`throws` ⟨*exception-class-name*⟩" to the header of the routine. For example:

```
/**
 * Returns the larger of the two roots of the quadratic equation
 * A*x*x + B*x + C = 0, provided it has any roots.  If A == 0 or
 * if the discriminant, B*B - 4*A*C, is negative, then an exception
 * of type IllegalArgumentException is thrown.
 */
static public double root( double A, double B, double C )
                                throws IllegalArgumentException {
    if (A == 0) {
      throw new IllegalArgumentException("A can't be zero.");
    }
    else {
       double disc = B*B - 4*A*C;
       if (disc < 0)
          throw new IllegalArgumentException("Discriminant < zero.");
       return  (-B + Math.sqrt(disc)) / (2*A);
    }
}
```

As discussed in the previous section, the computation in this subroutine has the preconditions that `A != 0` and `B*B-4*A*C >= 0`. The subroutine throws an exception of type *IllegalArgumentException* when either of these preconditions is violated. When an illegal condition is found in a subroutine, throwing an exception is often a reasonable response. If the program that called the subroutine knows some good way to handle the error, it can catch the exception. If not, the program will crash—and the programmer will know that the program needs to be fixed.

A `throws` clause in a subroutine heading can declare several different types of exception, separated by commas. For example:

```
void processArray(int[] A) throws NullPointerException,
                                        ArrayIndexOutOfBoundsException { ...
```

8.3.4 Mandatory Exception Handling

In the preceding example, declaring that the subroutine `root()` can throw an *IllegalArgumentException* is just a courtesy to potential readers of this routine. This is because handling of *IllegalArgumentExceptions* is not "mandatory." A routine can throw an *IllegalArgumentException* without announcing the possibility. And a program that calls that routine is free either to catch or to ignore the exception, just as a programmer can choose either to catch or to ignore an exception of type *NullPointerException*.

For those exception classes that require mandatory handling, the situation is different. If a subroutine can throw such an exception, that fact **must** be announced in a `throws` clause in the routine definition. Failing to do so is a syntax error that will be reported by the compiler. Exceptions that require mandatory handling are called ***checked exceptions***. The compiler will check that such exceptions are handled by the program.

Suppose that some statement in the body of a subroutine can generate a checked exception, one that requires mandatory handling. The statement could be a `throw` statement, which throws the exception directly, or it could be a call to a subroutine that can throw the exception. In either case, the exception **must** be handled. This can be done in one of two ways: The first way is to place the statement in a `try` statement that has a `catch` clause that handles the exception; in this case, the exception is handled within the subroutine, so that any caller of the subroutine will never see the exception. The second way is to declare that the subroutine can throw the exception. This is done by adding a "`throws`" clause to the subroutine heading, which alerts any callers to the possibility that an exception might be generated when the subroutine is executed. The caller will, in turn, be forced either to handle the exception in a `try` statement or to declare the exception in a `throws` clause in its own header.

Exception-handling is mandatory for any exception class that is not a subclass of either *Error* or *RuntimeException*. These checked exceptions generally represent conditions that are outside the control of the programmer. For example, they might represent bad input or an illegal action taken by the user. There is no way to **avoid** such errors, so a robust program has to be prepared to handle them. The design of Java makes it impossible for programmers to ignore the possibility of such errors.

Among the checked exceptions are several that can occur when using Java's input/output routines. This means that you can't even use these routines unless you understand something about exception-handling. Chapter 11 deals with input/output and uses mandatory exception-handling extensively.

8.3.5 Programming with Exceptions

Exceptions can be used to help write robust programs. They provide an organized and structured approach to robustness. Without exceptions, a program can become cluttered with `if` statements that test for various possible error conditions. With exceptions, it becomes possible to write a clean implementation of an algorithm that will handle all the normal cases. The exceptional cases can be handled elsewhere, in a `catch` clause of a `try` statement.

When a program encounters an exceptional condition and has no way of handling it immediately, the program can throw an exception. In some cases, it makes sense to throw an exception belonging to one of Java's predefined classes, such as *IllegalArgumentException* or *IOException*. However, if there is no standard class that adequately represents the exceptional condition, the programmer can define a new exception class. The new class must extend the standard class *Throwable* or one of its subclasses. In general, if the programmer does **not** want to require mandatory exception handling, the new class will extend *RuntimeException* (or one of its subclasses). To create a new checked exception class, which **does** require mandatory handling, the programmer can extend one of the other subclasses of *Exception* or can extend *Exception* itself.

Here, for example, is a class that extends *Exception*, and therefore requires mandatory exception handling when it is used:

```
public class ParseError extends Exception {
   public ParseError(String message) {
         // Create a ParseError object containing
         // the given message as its error message.
      super(message);
   }
}
```

The class contains only a constructor that makes it possible to create a *ParseError* object containing a given error message. (The statement "`super(message)`" calls a constructor in the superclass, *Exception*. See Subsection 5.6.3.) Of course the class inherits the `getMessage()` and `printStackTrace()` routines from its superclass. If `e` refers to an object of type `ParseError`, then the function call `e.getMessage()` will retrieve the error message that was specified in the constructor. But the main point of the *ParseError* class is simply to exist. When an object of type *ParseError* is thrown, it indicates that a certain type of error has occurred. (***Parsing***, by the way, refers to figuring out the syntax of a string. A *ParseError* would indicate, presumably, that some string that is being processed by the program does not have the expected form.)

A `throw` statement can be used in a program to throw an error of type *ParseError*. The constructor for the *ParseError* object must specify an error message. For example:

```
throw new ParseError("Encountered an illegal negative number.");
```

or

```
throw new ParseError("The word '" + word
                                + "' is not a valid file name.");
```

If the `throw` statement does not occur in a `try` statement that catches the error, then the subroutine that contains the `throw` statement must declare that it can throw a *ParseError* by adding the clause "`throws ParseError`" to the subroutine heading. For example,

```
void getUserData() throws ParseError {
   . . .
}
```

This would not be required if *ParseError* were defined as a subclass of *RuntimeException* instead of *Exception*, since in that case *ParseErrors* would not be checked exceptions.

A routine that wants to handle *ParseErrors* can use a `try` statement with a `catch` clause that catches *ParseErrors*. For example:

```
try {
   getUserData();
   processUserData();
}
catch (ParseError pe) {
   . . .  // Handle the error
}
```

Note that since *ParseError* is a subclass of *Exception*, a `catch` clause of the form "`catch (Exception e)`" would also catch *ParseErrors*, along with any other object of type *Exception*.

Sometimes, it's useful to store extra data in an exception object. For example,

```
class ShipDestroyed extends RuntimeException {
   Ship ship;  // Which ship was destroyed.
   int where_x, where_y;  // Location where ship was destroyed.
   ShipDestroyed(String message, Ship s, int x, int y) {
         // Constructor creates a ShipDestroyed object
         // carrying an error message plus the information
         // that the ship s was destroyed at location (x,y)
         // on the screen.
       super(message);
       ship = s;
       where_x = x;
```

```
            where_y = y;
        }
    }
```

Here, a *ShipDestroyed* object contains an error message and some information about a ship that was destroyed. This could be used, for example, in a statement:

```
if ( userShip.isHit() )
   throw new ShipDestroyed("You've been hit!", userShip, xPos, yPos);
```

Note that the condition represented by a *ShipDestroyed* object might not even be considered an error. It could be just an expected interruption to the normal flow of a game. Exceptions can sometimes be used to handle such interruptions neatly.

* * *

The ability to throw exceptions is particularly useful in writing general-purpose methods and classes that are meant to be used in more than one program. In this case, the person writing the method or class often has no reasonable way of handling the error, since that person has no way of knowing exactly how the method or class will be used. In such circumstances, a novice programmer is often tempted to print an error message and forge ahead, but this is almost never satisfactory since it can lead to unpredictable results down the line. Printing an error message and terminating the program is almost as bad, since it gives the program no chance to handle the error.

The program that calls the method or uses the class needs to know that the error has occurred. In languages that do not support exceptions, the only alternative is to return some special value or to set the value of some variable to indicate that an error has occurred. For example, the `readMeasurement()` function in Subsection 8.2.2 returns the value `-1` if the user's input is illegal. However, this only does any good if the main program bothers to test the return value. It is very easy to be lazy about checking for special return values every time a subroutine is called. And in this case, using `-1` as a signal that an error has occurred makes it impossible to allow negative measurements. Exceptions are a cleaner way for a subroutine to react when it encounters an error.

It is easy to modify the `readMeasurement()` function to use exceptions instead of a special return value to signal an error. My modified subroutine throws a *ParseError* when the user's input is illegal, where *ParseError* is the subclass of *Exception* that was defined above. (Arguably, it might be reasonable to avoid defining a new class by using the standard exception class *IllegalArgumentException* instead.) The changes from the original version are shown in italic:

```
/**
 * Reads the user's input measurement from one line of input.
 * Precondition:   The input line is not empty.
 * Postcondition:  If the user's input is legal, the measurement
 *                 is converted to inches and returned.
 * @throws ParseError if the user's input is not legal.
 */
static double readMeasurement() throws ParseError {

   double inches;  // Total number of inches in user's measurement.

   double measurement;  // One measurement,
                        //   such as the 12 in "12 miles."
   String units;        // The units specified for the measurement,
                        //   such as "miles."
```

```
char ch;  // Used to peek at next character in the user's input.

inches = 0;  // No inches have yet been read.

skipBlanks();
ch = TextIO.peek();

/* As long as there is more input on the line, read a measurement and
   add the equivalent number of inches to the variable, inches.  If an
   error is detected during the loop, end the subroutine immediately
   by throwing a ParseError. */

while (ch != '\n') {

    /* Get the next measurement and the units.  Before reading
       anything, make sure that a legal value is there to read. */

    if ( ! Character.isDigit(ch) ) {
        throw new ParseError("Expected to find a number, but found " + ch);
    }
    measurement = TextIO.getDouble();

    skipBlanks();
    if (TextIO.peek() == '\n') {
       throw new ParseError("Missing unit of measure at end of line.");
    }
    units = TextIO.getWord();
    units = units.toLowerCase();

    /* Convert the measurement to inches and add it to the total. */

    if (units.equals("inch")
            || units.equals("inches") || units.equals("in")) {
        inches += measurement;
    }
    else if (units.equals("foot")
               || units.equals("feet") || units.equals("ft")) {
        inches += measurement * 12;
    }
    else if (units.equals("yard")
               || units.equals("yards") || units.equals("yd")) {
        inches += measurement * 36;
    }
    else if (units.equals("mile")
               || units.equals("miles") || units.equals("mi")) {
        inches += measurement * 12 * 5280;
    }
    else {
        throw new ParseError("\"" + units
                             + "\" is not a legal unit of measure.");
    }

    /* Look ahead to see whether the next thing on the line is
       the end-of-line. */

    skipBlanks();
    ch = TextIO.peek();

}  // end while
```

```
        return inches;

    } // end readMeasurement()
```

In the main program, this subroutine is called in a `try` statement of the form

```
try {
   inches = readMeasurement();
}
catch (ParseError e) {
   . . .  // Handle the error.
}
```

The complete program can be found in the file *LengthConverter3.java*. From the user's point of view, this program has exactly the same behavior as the program *LengthConverter2* from the previous section. Internally, however, the programs are significantly different, since `LengthConverter3` uses exception handling.

8.4 Assertions and Annotations

In this short section, we look briefly at two features of Java that are not covered or used elsewhere in this textbook, assertions and annotations. They are included here for completeness, but they are mostly meant for more advanced programming.

8.4.1 Assertions

Recall that a precondition is a condition that must be true at a certain point in a program, for the execution of the program to continue correctly from that point. In the case where there is a chance that the precondition might not be satisfied—for example, if it depends on input from the user—then it's a good idea to insert an `if` statement to test it. But then the question arises, What should be done if the precondition does not hold? One option is to throw an exception. This will terminate the program, unless the exception is caught and handled elsewhere in the program.

In many cases, of course, instead of using an `if` statement to *test* whether a precondition holds, a programmer tries to write the program in a way that will *guarantee* that the precondition holds. In that case, the test should not be necessary, and the `if` statement can be avoided. The problem is that programmers are not perfect. In spite of the programmer's intention, the program might contain a bug that screws up the precondition. So maybe it's a good idea to check the precondition after all—at least during the debugging phase of program development.

Similarly, a postcondition is a condition that is true at a certain point in the program as a consequence of the code that has been executed before that point. Assuming that the code is correctly written, a postcondition is guaranteed to be true, but here again testing whether a desired postcondition is **actually** true is a way of checking for a bug that might have screwed up the postcondition. This is something that might be desirable during debugging.

The programming languages C and C++ have always had a facility for adding what are called ***assertions*** to a program. These assertions take the form "`assert(`⟨*condition*⟩`)`", where ⟨*condition*⟩ is a **boolean**-valued expression. This condition expresses a precondition or postcondition that should hold at that point in the program. When the computer encounters an assertion during the execution of the program, it evaluates the condition. If the condition is false, the program is terminated. Otherwise, the program continues normally. This allows the

programmer's belief that the condition is true to be tested; if it is not true, that indicates that the part of the program that preceded the assertion contained a bug. One nice thing about assertions in C and C++ is that they can be "turned off" at compile time. That is, if the program is compiled in one way, then the assertions are included in the compiled code. If the program is compiled in another way, the assertions are not included. During debugging, the first type of compilation is used, with assertions turned on. The release version of the program is compiled with assertions turned off. The release version will be more efficient, because the computer won't have to evaluate all the assertions.

Although early versions of Java did not have assertions, an assertion facility similar to the one in C/C++ has been available in Java since version 1.4. As with the C/C++ version, Java assertions can be turned on during debugging and turned off during normal execution. In Java, however, assertions are turned on and off at run time rather than at compile time. An assertion in the Java source code is always included in the compiled class file. When the program is run in the normal way, these assertions are ignored; since the condition in the assertion is not evaluated in this case, there is little or no performance penalty for having the assertions in the program. When the program is being debugged, it can be run with assertions enabled, as discussed below, and then the assertions can be a great help in locating and identifying bugs.

* * *

An ***assertion statement*** in Java takes one of the following two forms:

```
assert ⟨condition⟩ ;
```

or

```
assert ⟨condition⟩ : ⟨error-message⟩ ;
```

where ⟨*condition*⟩ is a **boolean**-valued expression and ⟨*error-message*⟩ is a string or an expression of type *String*. The word "`assert`" is a reserved word in Java, which cannot be used as an identifier. An assertion statement can be used anyplace in Java where a statement is legal.

If a program is run with assertions disabled, an assertion statement is equivalent to an empty statement and has no effect. When assertions are enabled and an assertion statement is encountered in the program, the ⟨*condition*⟩ in the assertion is evaluated. If the value is `true`, the program proceeds normally. If the value of the condition is `false`, then an exception of type `java.lang.AssertionError` is thrown, and the program will crash (unless the error is caught by a `try` statement). If the `assert` statement includes an ⟨*error-message*⟩, then the error message string becomes the message in the *AssertionError*.

So, the statement "`assert` ⟨*condition*⟩ `:` ⟨*error-message*⟩`;`" is similar to

```
if ( ⟨condition⟩ == false )
    throw new AssertionError( ⟨error-message⟩ );
```

except that the `if` statement is executed whenever the program is run, and the `assert` statement is executed only when the program is run with assertions enabled.

The question is, when to use assertions instead of exceptions? The general rule is to use assertions to test conditions that should definitely be true, if the program is written correctly. Assertions are useful for testing a program to see whether or not it is correct and for finding the errors in an incorrect program. After testing and debugging, when the program is used in the normal way, the assertions in the program will be ignored. However, if a problem turns up later, the assertions are still there in the program to be used to help locate the error. If someone writes to you to say that your program doesn't work when he does such-and-such, you

can run the program with assertions enabled, do such-and-such, and hope that the assertions in the program will help you locate the point in the program where it goes wrong.

Consider, for example, the `root()` method from Subsection 8.3.3 that calculates a root of a quadratic equation. If you believe that your program will always call this method with legal arguments, then it would make sense to write the method using assertions instead of exceptions:

```
/**
 * Returns the larger of the two roots of the quadratic equation
 * A*x*x + B*x + C = 0, provided it has any roots.
 * Precondition: A != 0 and B*B - 4*A*C >= 0.
 */
static public double root( double A, double B, double C )  {
   assert A != 0 : "Leading coefficient of quadratic equation cannot be zero.";
   double disc = B*B - 4*A*C;
   assert disc >= 0 : "Discriminant of quadratic equation cannot be negative.";
   return  (-B + Math.sqrt(disc)) / (2*A);
}
```

The assertions are not checked when the program is run in the normal way. If you are correct in your belief that the method is never called with illegal arguments, then checking the conditions in the assertions would be unnecessary. If your belief is not correct, the problem should turn up during testing or debugging, when the program is run with the assertions enabled.

If the `root()` method is part of a software library that you expect other people to use, then the situation is less clear. Oracle's Java documentation advises that assertions should **not** be used for checking the contract of public methods: If the caller of a method violates the contract by passing illegal parameters, then an exception should be thrown. This will enforce the contract whether or not assertions are enabled. (However, while it's true that Java programmers *expect* the contract of a method to be enforced with exceptions, there are reasonable arguments for using assertions instead, in some cases.) One might say that assertions are for **you**, to help you in debugging your code, while exceptions are for people who use your code, to alert them that they are misusing it.

On the other hand, it never hurts to use an assertion to check a postcondition of a method. A postcondition is something that is supposed to be true after the method has executed, and it can be tested with an `assert` statement at the end of the method. If the postcondition is false, there is a bug in the method itself, and that is something that needs to be found during the development of the method.

* * *

To have any effect, assertions must be **enabled** when the program is run. How to do this depends on what programming environment you are using. (See Section 2.6 for a discussion of programming environments.) In the usual command line environment, assertions are enabled by adding the option `-enableassertions` to the `java` command that is used to run the program. For example, if the class that contains the main program is *RootFinder*, then the command

```
java -enableassertions RootFinder
```

will run the program with assertions enabled. The `-enableassertions` option can be abbreviated to `-ea`, so the command can alternatively be written as

```
java -ea RootFinder
```

In fact, it is possible to enable assertions in just part of a program. An option of the form "`-ea:`⟨*class-name*⟩" enables only the assertions in the specified class. Note that there are no

spaces between the `-ea`, the ":", and the name of the class. To enable all the assertions in a package and in its sub-packages, you can use an option of the form "`-ea:`⟨*package-name*⟩`...`". To enable assertions in the "default package" (that is, classes that are not specified to belong to a package, like almost all the classes in this book), use "`-ea:...`". For example, to run a Java program named "MegaPaint" with assertions enabled for every class in the packages named "paintutils" and "drawing", you would use the command:

```
java  -ea:paintutils...  -ea:drawing...  MegaPaint
```

If you are using the Eclipse integrated development environment, you can specify the `-ea` option by creating a ***run configuration***. Right-click the name of the main program class in the Package Explorer pane, and select "Run As" from the pop-up menu and then "Run..." from the submenu. This will open a dialog box where you can manage run configurations. The name of the project and of the main class will be already be filled in. Click the "Arguments" tab, and enter `-ea` in the box under "VM Arguments". The contents of this box are added to the `java` command that is used to run the program. You can enter other options in this box, including more complicated `enableassertions` options such as `-ea:paintutils...`. When you click the "Run" button, the options will be applied. Furthermore, they will be applied whenever you run the program, unless you change the run configuration or add a new configuration. Note that it is possible to make two run configurations for the same class, one with assertions enabled and one with assertions disabled.

8.4.2 Annotations

The term "annotation" refers to notes added to or written alongside a main text, to help you understand or appreciate the text. An annotation might be a note that you make to yourself in the margin of a book. It might be a footnote added to an old novel by an editor to explain the historical context of some event. The annotation is metadata or "metatext," that is, text written *about* the main text rather than as *part of* the main text itself.

Comments on a program are actually a kind of annotation. Since they are ignored by the compiler, they have no effect on the meaning of the program. They are there to explain that meaning to a human reader. It is possible, of course, for another computer program (not the compiler) to process comments. That's what done in the case of Javadoc comments, which are processed by a program that uses them to create API documentation. But comments are only one type of metadata that might be added to programs.

In Java 5.0, a new feature called ***annotations*** was added to the Java language to make it easier to create new kinds of metadata for Java programs. This has made it possible for programmers to devise new ways of annotating programs, and to write programs that can read and use their annotations.

Java annotations have no direct effect on the program that they annotate. But they do have many potential uses. Some annotations are used to make the programmer's intent more explicit. Such annotations might be checked by a compiler to make sure that the code is consistent with the programmer's intention. For example, `@Override` is a standard annotation that can be used to annotate method definitions. It means that the method is intended to override (that is replace) a method with the same signature that was defined in some superclass. A compiler can check that the superclass method actually exists; if not, it can inform the programmer. An annotation used in this way is an aid to writing correct programs, since the programmer can be warned about a potential error in advance, instead of having to hunt it down later as a bug.

To annotate a method definition with the `@Override` annotation, simply place it in front of the definition. Syntactically, annotations are modifiers that are used in much the same way as built-in modifiers like "public" and "final." For example,

```
@Override public void WindowClosed(WindowEvent evt) { ... }
```

If there is no `"WindowClosed(WindowEvent)"` method in any superclass, then the compiler can issue an error. In fact, this example is based on a hard-to-find bug that I once introduced when trying to override a method named "windowClosed" with a method that I called "WindowClosed" (with an upper case "W"). If the `@Override` annotation had existed at that time—and if I had used it—the compiler would have rejected my code and saved me the trouble of tracking down the bug.

(Annotations are a fairly advanced feature, and I might not have mentioned them in this textbook, except that the `@Override` annotation can show up in code generated by Eclipse and other integrated development environments.)

There are two other standard annotations. One is `@Deprecated`, which can be used to mark deprecated classes, methods, and variables. (A deprecated item is one that is considered to be obsolete, but is still part of the Java language for backwards compatibility for old code.) Use of this annotation would allow a compiler to generate warnings when the deprecated item is used.

The other standard annotation is `@SurpressWarnings`, which can be used by a compiler to turn off warning messages that would ordinarily be generated when a class or method is compiled. `@SuppressWarnings` is an example of an annotation that has a parameter. The parameter tells what class of warnings are to be suppressed. For example, when a class or method is annotated with

```
@SuppressWarnings("deprecation")
```

then no warnings about the use of deprecated items will be emitted when the class or method is compiled. There are other types of warning that can be suppressed; unfortunately the list of warnings and their names is not standardized and will vary from one compiler to another.

Note, by the way, that the syntax for annotation parameters—especially for an annotation that accepts multiple parameters—is not the same as the syntax for method parameters. I won't cover the annotation syntax here.

Programmers can define new annotations for use in their code. Such annotations are ignored by standard compilers and programming tools, but it's possible to write programs that can understand the annotations and check for their presence in source code. It is even possible to create annotations that will be retained at run-time and become part of the running program. In that case, a program can check for annotations in the actual compiled code that is being executed, and take actions that depend on the presence of the annotation or the values of its parameters.

Annotations can help programmers to write correct programs. To use an example from the Java documentation, they can help with the creation of "boilerplate" code—that is, code that has a very standardized format and that can be generated mechanically. Often, boilerplate code is generated based on other code. Doing that by hand is a tedious and error-prone process. A simple example might be code to save certain aspects of a program's state to a file and to restore it later. The code for reading and writing the values of all the relevant state variables is highly repetitious. Instead of writing that code by hand, a programmer could use an annotation to mark the variables that are part of the state that is to be saved. A program could then be used to check for the annotations and generate the save-and-restore code. In fact, it would even

be possible to do without that code altogether, if the program checks for the presence of the annotation at run time to decide which variables to save and restore.

8.5 Analysis of Algorithms

THIS CHAPTER HAS CONCENTRATED mostly on correctness of programs. In practice, another issue is also important: ***efficiency***. When analyzing a program in terms of efficiency, we want to look at questions such as, "How long does it take for the program to run?" and "Is there another approach that will get the answer more quickly?" Efficiency will always be less important than correctness; if you don't care whether a program works correctly, you can make it run very quickly indeed, but no one will think it's much of an achievement! On the other hand, a program that gives a correct answer after ten thousand years isn't very useful either, so efficiency is often an important issue.

The term "efficiency" can refer to efficient use of almost any resource, including time, computer memory, disk space, or network bandwidth. In this section, however, we will deal exclusively with time efficiency, and the major question that we want to ask about a program is, how long does it take to perform its task?

It really makes little sense to classify an individual program as being "efficient" or "inefficient." It makes more sense to compare two (correct) programs that perform the same task and ask which one of the two is "more efficient," that is, which one performs the task more quickly. However, even here there are difficulties. The running time of a program is not well-defined. The run time can be different depending on the number and speed of the processors in the computer on which it is run and, in the case of Java, on the design of the Java Virtual Machine which is used to interpret the program. It can depend on details of the compiler which is used to translate the program from high-level language to machine language. Furthermore, the run time of a program depends on the size of the problem which the program has to solve. It takes a sorting program longer to sort 10000 items than it takes it to sort 100 items. When the run times of two programs are compared, it often happens that Program A solves small problems faster than Program B, while Program B solves large problems faster than Program A, so that it is simply not the case that one program is faster than the other in all cases.

In spite of these difficulties, there is a field of computer science dedicated to analyzing the efficiency of programs. The field is known as ***Analysis of Algorithms***. The focus is on algorithms, rather than on programs as such, to avoid having to deal with multiple implementations of the same algorithm written in different languages, compiled with different compilers, and running on different computers. Analysis of Algorithms is a mathematical field that abstracts away from these down-and-dirty details. Still, even though it is a theoretical field, every working programmer should be aware of some of its techniques and results. This section is a very brief introduction to some of those techniques and results. Because this is not a mathematics book, the treatment will be rather informal.

One of the main techniques of analysis of algorithms is ***asymptotic analysis***. The term "asymptotic" here means basically "the tendency in the long run." An asymptotic analysis of an algorithm's run time looks at the question of how the run time depends on the size of the problem. The analysis is asymptotic because it only considers what happens to the run time as the size of the problem increases without limit; it is not concerned with what happens for problems of small size or, in fact, for problems of any fixed finite size. Only what happens in the long run, as the problem size increases without limit, is important. Showing that Algorithm A is asymptotically faster than Algorithm B doesn't necessarily mean that Algorithm A will run

faster than Algorithm B for problems of size 10 or size 1000 or even size 1000000—it only means that if you keep increasing the problem size, you will eventually come to a point where Algorithm A is faster than Algorithm B. An asymptotic analysis is only a first approximation, but in practice it often gives important and useful information.

* * *

Central to asymptotic analysis is ***Big-Oh notation***. Using this notation, we might say, for example, that an algorithm has a running time that is $\mathcal{O}(n^2)$ or $\mathcal{O}(n)$ or $\mathcal{O}(\log(n))$. These notations are read "Big-Oh of n squared," "Big-Oh of n," and "Big-Oh of log n" (where log is a logarithm function). More generally, we can refer to $\mathcal{O}(f(n))$ ("Big-Oh of f of n"), where f(n) is some function that assigns a positive real number to every positive integer n. The "n" in this notation refers to the size of the problem. Before you can even begin an asymptotic analysis, you need some way to measure problem size. Usually, this is not a big issue. For example, if the problem is to sort a list of items, then the problem size can be taken to be the number of items in the list. When the input to an algorithm is an integer, as in the case of an algorithm that checks whether a given positive integer is prime, the usual measure of the size of a problem is the number of bits in the input integer rather than the integer itself. More generally, the number of bits in the input to a problem is often a good measure of the size of the problem.

To say that the running time of an algorithm is $\mathcal{O}(f(n))$ means that for large values of the problem size, n, the running time of the algorithm is no bigger than some constant times f(n). (More rigorously, there is a number C and a positive integer M such that whenever n is greater than M, the run time is less than or equal to C*f(n).) The constant takes into account details such as the speed of the computer on which the algorithm is run; if you use a slower computer, you might have to use a bigger constant in the formula, but changing the constant won't change the basic fact that the run time is $\mathcal{O}(f(n))$. The constant also makes it unnecessary to say whether we are measuring time in seconds, years, CPU cycles, or any other unit of measure; a change from one unit of measure to another is just multiplication by a constant. Note also that $\mathcal{O}(f(n))$ doesn't depend at all on what happens for small problem sizes, only on what happens in the long run as the problem size increases without limit.

To look at a simple example, consider the problem of adding up all the numbers in an array. The problem size, n, is the length of the array. Using `A` as the name of the array, the algorithm can be expressed in Java as:

```
total = 0;
for (int i = 0; i < n; i++)
   total = total + A[i];
```

This algorithm performs the same operation, `total = total + A[i]`, n times. The total time spent on this operation is a*n, where a is the time it takes to perform the operation once. Now, this is not the only thing that is done in the algorithm. The value of `i` is incremented and is compared to n each time through the loop. This adds an additional time of b*n to the run time, for some constant b. Furthermore, `i` and `total` both have to be initialized to zero; this adds some constant amount c to the running time. The exact running time would then be (a+b)*n+c, where the constants a, b, and c depend on factors such as how the code is compiled and what computer it is run on. Using the fact that c is less than or equal to c*n for any positive integer n, we can say that the run time is less than or equal to (a+b+c)*n. That is, the run time is less than or equal to a constant times n. By definition, this means that the run time for this algorithm is $\mathcal{O}(n)$.

If this explanation is too mathematical for you, we can just note that for large values of n, the c in the formula (a+b)*n+c is insignificant compared to the other term, (a+b)*n. We

say that c is a "lower order term." When doing asymptotic analysis, lower order terms can be discarded. A rough, but correct, asymptotic analysis of the algorithm would go something like this: Each iteration of the `for` loop takes a certain constant amount of time. There are n iterations of the loop, so the total run time is a constant times n, plus lower order terms (to account for the initialization). Disregarding lower order terms, we see that the run time is $\mathcal{O}(n)$.

* * *

Note that to say that an algorithm has run time $\mathcal{O}(f(n))$ is to say that its run time is no bigger than some constant times f(n) (for large values of n). $\mathcal{O}(f(n))$ puts an **upper limit** on the run time. However, the run time could be smaller, even much smaller. For example, if the run time is $\mathcal{O}(n)$, it would also be correct to say that the run time is $\mathcal{O}(n^2)$ or even $\mathcal{O}(n^{10})$. If the run time is less than a constant times n, then it is certainly less than the same constant times n^2 or n^{10}.

Of course, sometimes it's useful to have a **lower limit** on the run time. That is, we want to be able to say that the run time is greater than or equal to some constant times f(n) (for large values of n). The notation for this is $\Omega(f(n))$, read "Omega of f of n." "Omega" is the name of a letter in the Greek alphabet, and Ω is the upper case version of that letter. (To be technical, saying that the run time of an algorithm is $\Omega(f(n))$ means that there is a positive number C and a positive integer M such that whenever n is greater than M, the run time is greater than or equal to C*f(n).) $\mathcal{O}(f(n))$ tells you something about the maximum amount of time that you might have to wait for an algorithm to finish; $\Omega(f(n))$ tells you something about the minimum time.

The algorithm for adding up the numbers in an array has a run time that is $\Omega(n)$ as well as $\mathcal{O}(n)$. When an algorithm has a run time that is both $\Omega(f(n))$ and $\mathcal{O}(f(n))$, its run time is said to be $\Theta(f(n))$, read "Theta of f of n." (Theta is another letter from the Greek alphabet.) To say that the run time of an algorithm is $\Theta(f(n))$ means that for large values of n, the run time is between a*f(n) and b*f(n), where a and b are constants (with b greater than a, and both greater than 0).

Let's look at another example. Consider the algorithm that can be expressed in Java in the following method:

```
/**
 * Sorts the n array elements A[0], A[1], ..., A[n-1] into increasing order.
 */
public static simpleBubbleSort( int[] A, int n ) {
   for (int i = 0; i < n; i++) {
         // Do n passes through the array...
      for (int j = 0; j < n-1; j++) {
         if ( A[j] > A[j+1] ) {
                // A[j] and A[j+1] are out of order, so swap them
             int temp = A[j];
             A[j] = A[j+1];
             A[j+1] = temp;
         }
      }
   }
}
```

Here, the parameter n represents the problem size. The outer `for` loop in the method is executed n times. Each time the outer `for` loop is executed, the inner `for` loop is executed n-1 times, so

the `if` statement is executed n*(n-1) times. This is n^2-n, but since lower order terms are not significant in an asymptotic analysis, it's good enough to say that the `if` statement is executed about n^2 times. In particular, the test `A[j] > A[j+1]` is executed about n^2 times, and this fact by itself is enough to say that the run time of the algorithm is $\Omega(n^2)$, that is, the run time is at least some constant times n^2. Furthermore, if we look at other operations—the assignment statements, incrementing `i` and `j`, etc.—none of them are executed more than n^2 times, so the run time is also $\mathcal{O}(n^2)$, that is, the run time is no more than some constant times n^2. Since it is both $\Omega(n^2)$ and $\mathcal{O}(n^2)$, the run time of the simpleBubbleSort algorithm is $\Theta(n^2)$.

You should be aware that some people use the notation $\mathcal{O}(f(n))$ as if it meant $\Theta(f(n))$. That is, when they say that the run time of an algorithm is $\mathcal{O}(f(n))$, they mean to say that the run time is about **equal** to a constant times f(n). For that, they should use $\Theta(f(n))$. Properly speaking, $\mathcal{O}(f(n))$ means that the run time is less than a constant times f(n), possibly much less.

* * *

So far, my analysis has ignored an important detail. We have looked at how run time depends on the problem size, but in fact the run time usually depends not just on the size of the problem but on the specific data that has to be processed. For example, the run time of a sorting algorithm can depend on the initial order of the items that are to be sorted, and not just on the number of items.

To account for this dependency, we can consider either the ***worst case*** run time analysis or the ***average case*** run time analysis of an algorithm. For a worst case run time analysis, we consider all possible problems of size n and look at the **longest** possible run time for all such problems. For an average case analysis, we consider all possible problems of size n and look at the **average** of the run times for all such problems. Usually, the average case analysis assumes that all problems of size n are equally likely to be encountered, although this is not always realistic—or even possible in the case where there is an infinite number of different problems of a given size.

In many cases, the average and the worst case run times are the same to within a constant multiple. This means that as far as asymptotic analysis is concerned, they are the same. That is, if the average case run time is $\mathcal{O}(f(n))$ or $\Theta(f(n))$, then so is the worst case. However, later in the book, we will encounter a few cases where the average and worst case asymptotic analyses differ.

* * *

So, what do you really have to know about analysis of algorithms to read the rest of this book? We will not do any rigorous mathematical analysis, but you should be able to follow informal discussion of simple cases such as the examples that we have looked at in this section. Most important, though, you should have a feeling for exactly what it means to say that the running time of an algorithm is $\mathcal{O}(f(n))$ or $\Theta(f(n))$ for some common functions f(n). The main point is that these notations do not tell you anything about the actual numerical value of the running time of the algorithm for any particular case. They do not tell you anything at all about the running time for small values of n. What they do tell you is something about the ***rate of growth*** of the running time as the size of the problem increases.

Suppose you compare two algorithms that solve the same problem. The run time of one algorithm is $\Theta(n^2)$, while the run time of the second algorithm is $\Theta(n^3)$. What does this tell you? If you want to know which algorithm will be faster for some particular problem of size, say, 100, nothing is certain. As far as you can tell just from the asymptotic analysis, either algorithm could be faster for that particular case—or in **any** particular case. But what you can

say for sure is that if you look at larger and larger problems, you will come to a point where the $\Theta(n^2)$ algorithm is faster than the $\Theta(n^3)$ algorithm. Furthermore, as you continue to increase the problem size, the relative advantage of the $\Theta(n^2)$ algorithm will continue to grow. There will be values of n for which the $\Theta(n^2)$ algorithm is a thousand times faster, a million times faster, a billion times faster, and so on. This is because for any positive constants a and b, the function $a*n^3$ **grows faster** than the function $b*n^2$ as n gets larger. (Mathematically, the limit of the ratio of $a*n^3$ to $b*n^2$ is infinite as n approaches infinity.)

This means that for "large" problems, a $\Theta(n^2)$ algorithm will definitely be faster than a $\Theta(n^3)$ algorithm. You just don't know—based on the asymptotic analysis alone—exactly how large "large" has to be. In practice, in fact, it is likely that the $\Theta(n^2)$ algorithm will be faster even for fairly small values of n, and absent other information you would generally prefer a $\Theta(n^2)$ algorithm to a $\Theta(n^3)$ algorithm.

So, to understand and apply asymptotic analysis, it is essential to have some idea of the rates of growth of some common functions. For the power functions n, n^2, n^3, n^4, ..., the larger the exponent, the greater the rate of growth of the function. Exponential functions such as 2^n and 10^n, where the n is in the exponent, have a growth rate that is faster than that of any power function. In fact, exponential functions grow so quickly that an algorithm whose run time grows exponentially is almost certainly impractical even for relatively modest values of n, because the running time is just too long. Another function that often turns up in asymptotic analysis is the logarithm function, log(n). There are actually many different logarithm functions, but the one that is usually used in computer science is the so-called logarithm to the base two, which is defined by the fact that $\log(2^x) = x$ for any number x. (Usually, this function is written $\log_2(n)$, but I will leave out the subscript 2, since I will only use the base-two logarithm in this book.) The logarithm function grows very slowly. The growth rate of log(n) is much smaller than the growth rate of n. The growth rate of n*log(n) is a little larger than the growth rate of n, but much smaller than the growth rate of n^2. The following table should help you understand the differences among the rates of grows of various functions:

n	log(n)	n*log(n)	n^2	n / log(n)
16	4	64	256	4.0
64	6	384	4096	10.7
256	8	2048	65536	32.0
1024	10	10240	1048576	102.4
1000000	20	19931568	1000000000000	50173.1
1000000000	30	29897352854	1000000000000000000	33447777.3

The reason that log(n) shows up so often is because of its association with multiplying and dividing by two: Suppose you start with the number n and divide it by 2, then divide by 2 again, and so on, until you get a number that is less than or equal to 1. Then the number of divisions is equal (to the nearest integer) to log(n).

As an example, consider the binary search algorithm from Subsection 7.4.1. This algorithm searches for an item in a sorted array. The problem size, n, can be taken to be the length of the array. Each step in the binary search algorithm divides the number of items still under consideration by 2, and the algorithm stops when the number of items under consideration is less than or equal to 1 (or sooner). It follows that the number of steps for an array of length n is at most log(n). This means that the worst-case run time for binary search is $\Theta(\log(n))$. (The average case run time is also $\Theta(\log(n))$.) By comparison, the linear search algorithm, which was also presented in Subsection 7.4.1 has a run time that is $\Theta(n)$. The Θ notation gives us

a quantitative way to express and to understand the fact that binary search is "much faster" than linear search.

In binary search, each step of the algorithm divides the problem size by 2. It often happens that some operation in an algorithm (not necessarily a single step) divides the problem size by 2. Whenever that happens, the logarithm function is likely to show up in an asymptotic analysis of the run time of the algorithm.

Analysis of Algorithms is a large, fascinating field. We will only use a few of the most basic ideas from this field, but even those can be very helpful for understanding the differences among algorithms.

Exercises for Chapter 8

1. Write a program that uses the following subroutine, from Subsection 8.3.3, to solve equations specified by the user.

```
/**
 * Returns the larger of the two roots of the quadratic equation
 * A*x*x + B*x + C = 0, provided it has any roots.  If A == 0 or
 * if the discriminant, B*B - 4*A*C, is negative, then an exception
 * of type IllegalArgumentException is thrown.
 */
static public double root( double A, double B, double C )
                                 throws IllegalArgumentException {
    if (A == 0) {
      throw new IllegalArgumentException("A can't be zero.");
    }
    else {
       double disc = B*B - 4*A*C;
       if (disc < 0)
          throw new IllegalArgumentException("Discriminant < zero.");
       return  (-B + Math.sqrt(disc)) / (2*A);
    }
}
```

Your program should allow the user to specify values for `A`, `B`, and `C`. It should call the subroutine to compute a solution of the equation. If no error occurs, it should print the root. However, if an error occurs, your program should catch that error and print an error message. After processing one equation, the program should ask whether the user wants to enter another equation. The program should continue until the user answers no.

2. As discussed in Section 8.1, values of type **int** are limited to 32 bits. Integers that are too large to be represented in 32 bits cannot be stored in an **int** variable. Java has a standard class, `java.math.BigInteger`, that addresses this problem. An object of type *BigInteger* is an integer that can be arbitrarily large. (The maximum size is limited only by the amount of memory available to the Java Virtual Machine.) Since *BigIntegers* are objects, they must be manipulated using instance methods from the *BigInteger* class. For example, you can't add two *BigIntegers* with the `+` operator. Instead, if `N` and `M` are variables that refer to *BigIntegers*, you can compute the sum of `N` and `M` with the function call `N.add(M)`. The value returned by this function is a new *BigInteger* object that is equal to the sum of `N` and `M`.

The `BigInteger` class has a constructor `new BigInteger(str)`, where `str` is a string. The string must represent an integer, such as "3" or "39849823783783283733". If the string does not represent a legal integer, then the constructor throws a *NumberFormatException*.

There are many instance methods in the *BigInteger* class. Here are a few that you will find useful for this exercise. Assume that `N` and `M` are variables of type `BigInteger`.

- `N.add(M)` — a function that returns a *BigInteger* representing the sum of `N` and `M`.
- `N.multiply(M)` — a function that returns a *BigInteger* representing the result of multiplying `N` times `M`.

- `N.divide(M)` — a function that returns a *BigInteger* representing the result of dividing `N` by `M`, discarding the remainder.
- `N.signum()` — a function that returns an ordinary **int**. The returned value represents the sign of the integer `N`. The returned value is 1 if `N` is greater than zero. It is -1 if `N` is less than zero. And it is 0 if `N` is zero.
- `N.equals(M)` — a function that returns a **boolean** value that is `true` if `N` and `M` have the same integer value.
- `N.toString()` — a function that returns a *String* representing the value of `N`.
- `N.testBit(k)` — a function that returns a **boolean** value. The parameter `k` is an integer. The return value is `true` if the `k`-th bit in `N` is 1, and it is `false` if the `k`-th bit is 0. Bits are numbered from right to left, starting with 0. Testing "`if (N.testBit(0))`" is an easy way to check whether `N` is even or odd. `N.testBit(0)` is `true` if and only if `N` is an odd number.

For this exercise, you should write a program that prints `3N+1` sequences with starting values specified by the user. In this version of the program, you should use *BigIntegers* to represent the terms in the sequence. You can read the user's input into a *String* with the `TextIO.getln()` function. Use the input value to create the *BigInteger* object that represents the starting point of the `3N+1` sequence. Don't forget to catch and handle the *NumberFormatException* that will occur if the user's input is not a legal integer! You should also check that the input number is greater than zero.

If the user's input is legal, print out the `3N+1` sequence. Count the number of terms in the sequence, and print the count at the end of the sequence. Exit the program when the user inputs an empty line.

3. A Roman numeral represents an integer using letters. Examples are XVII to represent 17, MCMLIII for 1953, and MMMCCCIII for 3303. By contrast, ordinary numbers such as 17 or 1953 are called Arabic numerals. The following table shows the Arabic equivalent of all the single-letter Roman numerals:

```
M    1000          X   10
D     500          V    5
C     100          I    1
L      50
```

When letters are strung together, the values of the letters are just added up, with the following exception. When a letter of smaller value is followed by a letter of larger value, the smaller value is subtracted from the larger value. For example, IV represents 5 - 1, or 4. And MCMXCV is interpreted as M + CM + XC + V, or 1000 + (1000 - 100) + (100 - 10) + 5, which is 1995. In standard Roman numerals, no more than three consecutive copies of the same letter are used. Following these rules, every number between 1 and 3999 can be represented as a Roman numeral made up of the following one- and two-letter combinations:

```
M    1000          X   10
CM    900          IX   9
D     500          V    5
CD    400          IV   4
C     100          I    1
XC     90
```

```
L      50
XL     40
```

Write a class to represent Roman numerals. The class should have two constructors. One constructs a Roman numeral from a string such as "XVII" or "MCMXCV". It should throw a *NumberFormatException* if the string is not a legal Roman numeral. The other constructor constructs a Roman numeral from an **int**. It should throw a *NumberFormatException* if the **int** is outside the range 1 to 3999.

In addition, the class should have two instance methods. The method `toString()` returns the string that represents the Roman numeral. The method `toInt()` returns the value of the Roman numeral as an **int**.

At some point in your class, you will have to convert an **int** into the string that represents the corresponding Roman numeral. One way to approach this is to gradually "move" value from the Arabic numeral to the Roman numeral. Here is the beginning of a routine that will do this, where `number` is the **int** that is to be converted:

```
String roman = "";
int N = number;
while (N >= 1000) {
      // Move 1000 from N to roman.
   roman += "M";
   N -= 1000;
}
while (N >= 900) {
      // Move 900 from N to roman.
   roman += "CM";
   N -= 900;
}
.
.  // Continue with other values from the above table.
.
```

(You can save yourself a lot of typing in this routine if you use arrays in a clever way to represent the data in the above table.)

Once you've written your class, use it in a main program that will read both Arabic numerals and Roman numerals entered by the user. If the user enters an Arabic numeral, print the corresponding Roman numeral. If the user enters a Roman numeral, print the corresponding Arabic numeral. (You can tell the difference by using `TextIO.peek()` to peek at the first character in the user's input (see Subsection 8.2.2). If the first character is a digit, then the user's input is an Arabic numeral. Otherwise, it's a Roman numeral.) The program should end when the user inputs an empty line.

4. The source code file *Expr.java* defines a class, *Expr*, that can be used to represent mathematical expressions involving the variable `x`. The expression can use the operators `+`, `-`, `*`, `/`, and `^` (where `^` represents the operation of raising a number to a power). It can use mathematical functions such as `sin`, `cos`, `abs`, and `ln`. See the source code file for full details. The *Expr* class uses some advanced techniques which have not yet been covered in this textbook. However, the interface is easy to understand. It contains only a constructor and two public methods.

The constructor `new Expr(def)` creates an *Expr* object defined by a given expression. The parameter, `def`, is a string that contains the definition. For example,

`new Expr("x^2")` or `new Expr("sin(x)+3*x")`. If the parameter in the constructor call does not represent a legal expression, then the constructor throws an *IllegalArgumentException*. The message in the exception describes the error.

If `func` is a variable of type `Expr` and `num` is of type **double**, then `func.value(num)` is a function that returns the value of the expression when the number `num` is substituted for the variable `x` in the expression. For example, if `Expr` represents the expression `3*x+1`, then `func.value(5)` is `3*5+1`, or 16. If the expression is undefined for the specified value of `x`, then the special value `Double.NaN` is returned; no exception is thrown.

Finally, `func.toString()` returns the definition of the expression. This is just the string that was used in the constructor that created the expression object.

For this exercise, you should write a program that lets the user enter an expression. If the expression contains an error, print an error message. Otherwise, let the user enter some numerical values for the variable `x`. Print the value of the expression for each number that the user enters. However, if the expression is undefined for the specified value of `x`, print a message to that effect. You can use the **boolean**-valued function `Double.isNaN(val)` to check whether a number, `val`, is `Double.NaN`.

The user should be able to enter as many values of `x` as desired. After that, the user should be able to enter a new expression. In the on-line version of this exercise, there is an applet that simulates my solution, so that you can see how it works.

5. This exercise uses the class *Expr*, which was described in Exercise 8.4 and which is defined in the source code file *Expr.java*. For this exercise, you should write a GUI program that can graph a function, `f(x)`, whose definition is entered by the user. The program should have a text-input box where the user can enter an expression involving the variable `x`, such as `x^2` or `sin(x-3)/x`. This expression is the definition of the function. When the user presses return in the text input box, the program should use the contents of the text input box to construct an object of type *Expr*. If an error is found in the definition, then the program should display an error message. Otherwise, it should display a graph of the function. (Note: A `JTextField` generates an `ActionEvent` when the user presses return.)

The program will need a *JPanel* for displaying the graph. To keep things simple, this panel should represent a fixed region in the xy-plane, defined by `-5 <= x <= 5` and `-5 <= y <= 5`. To draw the graph, compute a large number of points and connect them with line segments. (This method does not handle discontinuous functions properly; doing so is very hard, so you shouldn't try to do it for this exercise.) My program divides the interval `-5 <= x <= 5` into 300 subintervals and uses the 301 endpoints of these subintervals for drawing the graph. Note that the function might be undefined at one of these `x`-values. In that case, you have to skip that point.

A point on the graph has the form `(x,y)` where `y` is obtained by evaluating the user's expression at the given value of `x`. You will have to convert these real numbers to the integer coordinates of the corresponding pixel on the canvas. The formulas for the conversion are:

```
a  =  (int)( (x + 5)/10 * width );
b  =  (int)( (5 - y)/10 * height );
```

where `a` and `b` are the horizontal and vertical coordinates of the pixel, and `width` and `height` are the width and height of the panel.

You can find an applet version of my solution in the on-line version of this exercise.

Quiz on Chapter 8

1. What does it mean to say that a program is *robust*?

2. Why do programming languages require that variables be declared before they are used? What does this have to do with correctness and robustness?

3. What is a *precondition*? Give an example.

4. Explain how preconditions can be used as an aid in writing correct programs.

5. Java has a predefined class called *Throwable*. What does this class represent? Why does it exist?

6. Write a method that prints out a `3N+1` sequence starting from a given integer, `N`. The starting value should be a parameter to the method. If the parameter is less than or equal to zero, throw an *IllegalArgumentException*. If the number in the sequence becomes too large to be represented as a value of type **int**, throw an *ArithmeticException*.

7. Rewrite the method from the previous question, using `assert` statements instead of exceptions to check for errors. What is the difference between the two versions of the method when the program is run?

8. Some classes of exceptions are *checked exceptions* that require *mandatory exception handling.* Explain what this means.

9. Consider a subroutine `processData()` that has the header

```
static void processData() throws IOException
```

Write a `try..catch` statement that calls this subroutine and prints an error message if an *IOException* occurs.

10. Why should a subroutine throw an exception when it encounters an error? Why not just terminate the program?

11. Suppose that you have a choice of two algorithms that perform the same task. One has average-case run time that is $\Theta(n^2)$ while the run time of the second algorithm has an average-case run time that is $\Theta(n*\log(n))$. Suppose that you need to process an input of size n = 100. Which algorithm would you choose? Can you be certain that you are choosing the fastest algorithm for the input that you intend to process.

12. Analyze the run time of the following algorithm. That is, find a function f(n) such that the run time of the algorithm is $\mathcal{O}(f(n))$ or, better, $\Theta(f(n))$. Assume that `A` is an array of integers, and use the length of the array as the input size, n.

```
int total = 0;
for (int i = 0; i < A.length; i++) {
   if (A[i] > 0)
      total = total + A[i];
}
```

Chapter 9

Linked Data Structures and Recursion

In this chapter, we look at two advanced programming techniques, recursion and linked data structures, and some of their applications. Both of these techniques are related to the seemingly paradoxical idea of defining something in terms of itself. This turns out to be a remarkably powerful idea.

A subroutine is said to be recursive if it calls itself, either directly or indirectly. What this means is that the subroutine is used in its own definition. Recursion can often be used to solve complex problems by reducing them to simpler problems of the same type.

A reference to one object can be stored in an instance variable of another object. The objects are then said to be "linked." Complex data structures can be built by linking objects together. An especially interesting case occurs when an object contains a link to another object that belongs to the same class. In that case, the class is used in its own definition. Several important types of data structures are built using classes of this kind.

9.1 Recursion

At one time or another, you've probably been told that you can't define something in terms of itself. Nevertheless, if it's done right, defining something at least partially in terms of itself can be a very powerful technique. A ***recursive*** definition is one that uses the concept or thing that is being defined as part of the definition. For example: An "ancestor" is either a parent or an *ancestor* of a parent. A "sentence" can be, among other things, two *sentences* joined by a conjunction such as "and." A "directory" is a part of a disk drive that can hold files and *directories*. In mathematics, a "set" is a collection of elements, which can themselves be *sets*. A "statement" in Java can be a `while` statement, which is made up of the word "while", a boolean-valued condition, and a *statement*.

Recursive definitions can describe very complex situations with just a few words. A definition of the term "ancestor" without using recursion might go something like "a parent, or a grandparent, or a great-grandparent, or a great-great-grandparent, and so on." But saying "and so on" is not very rigorous. (I've often thought that recursion is really just a rigorous way of saying "and so on.") You run into the same problem if you try to define a "directory" as "a file that is a list of files, where some of the files can be lists of files, where some of **those** files can be lists of files, and so on." Trying to describe what a Java statement can look like, without using recursion in the definition, would be difficult and probably pretty comical.

Recursion can be used as a programming technique. A ***recursive subroutine*** is one that calls itself, either directly or indirectly. To say that a subroutine calls itself directly means that its definition contains a subroutine call statement that calls the subroutine that is being defined. To say that a subroutine calls itself indirectly means that it calls a second subroutine which in turn calls the first subroutine (either directly or indirectly). A recursive subroutine can define a complex task in just a few lines of code. In the rest of this section, we'll look at a variety of examples, and we'll see other examples in the rest of the book.

9.1.1 Recursive Binary Search

Let's start with an example that you've seen before: the binary search algorithm from Subsection 7.4.1. Binary search is used to find a specified value in a sorted list of items (or, if it does not occur in the list, to determine that fact). The idea is to test the element in the middle of the list. If that element is equal to the specified value, you are done. If the specified value is less than the middle element of the list, then you should search for the value in the first half of the list. Otherwise, you should search for the value in the second half of the list. The method used to search for the value in the first or second half of the list is binary search. That is, you look at the middle element in the half of the list that is still under consideration, and either you've found the value you are looking for, or you have to apply binary search to one half of the remaining elements. And so on! This is a recursive description, and we can write a recursive subroutine to implement it.

Before we can do that, though, there are two considerations that we need to take into account. Each of these illustrates an important general fact about recursive subroutines. First of all, the binary search algorithm begins by looking at the "middle element of the list." But what if the list is empty? If there are no elements in the list, then it is impossible to look at the middle element. In the terminology of Subsection 8.2.1, having a non-empty list is a "precondition" for looking at the middle element, and this is a clue that we have to modify the algorithm to take this precondition into account. What should we do if we find ourselves searching for a specified value in an empty list? The answer is easy: If the list is empty, we can be sure that the value does not occur in the list, so we can give the answer without any further work. An empty list is a ***base case*** for the binary search algorithm. A base case for a recursive algorithm is a case that is handled directly, rather than by applying the algorithm recursively. The binary search algorithm actually has another type of base case: If we find the element we are looking for in the middle of the list, we are done. There is no need for further recursion.

The second consideration has to do with the parameters to the subroutine. The problem is phrased in terms of searching for a value in a list. In the original, non-recursive binary search subroutine, the list was given as an array. However, in the recursive approach, we have to be able to apply the subroutine recursively to just a **part** of the original list. Where the original subroutine was designed to search an entire array, the recursive subroutine must be able to search part of an array. The parameters to the subroutine must tell it what part of the array to search. This illustrates a general fact that in order to solve a problem recursively, it is often necessary to generalize the problem slightly.

Here is a recursive binary search algorithm that searches for a given value in part of an array of integers:

```
/**
 * Search in the array A in positions numbered loIndex to hiIndex,
 * inclusive, for the specified value.   If the value is found, return
 * the index in the array where it occurs. If the value is not found,
```

```
     * return -1.  Precondition:  The array must be sorted into increasing
     * order.
     */
    static int binarySearch(int[] A, int loIndex, int hiIndex, int value) {

       if (loIndex > hiIndex) {
             // The starting position comes after the final index,
             // so there are actually no elements in the specified
             // range.  The value does not occur in this empty list!
          return -1;
       }

       else {
             // Look at the middle position in the list.  If the
             // value occurs at that position, return that position.
             // Otherwise, search recursively in either the first
             // half or the second half of the list.
          int middle = (loIndex + hiIndex) / 2;
          if (value == A[middle])
             return middle;
          else if (value < A[middle])
             return binarySearch(A, loIndex, middle - 1, value);
          else   // value must be > A[middle]
             return binarySearch(A, middle + 1, hiIndex, value);
       }

    } // end binarySearch()
```

In this routine, the parameters `loIndex` and `hiIndex` specify the part of the array that is to be searched. To search an entire array, it is only necessary to call `binarySearch(A, 0, A.length - 1, value)`. In the two base cases—when there are no elements in the specified range of indices and when the value is found in the middle of the range—the subroutine can return an answer immediately, without using recursion. In the other cases, it uses a recursive call to compute the answer and returns that answer.

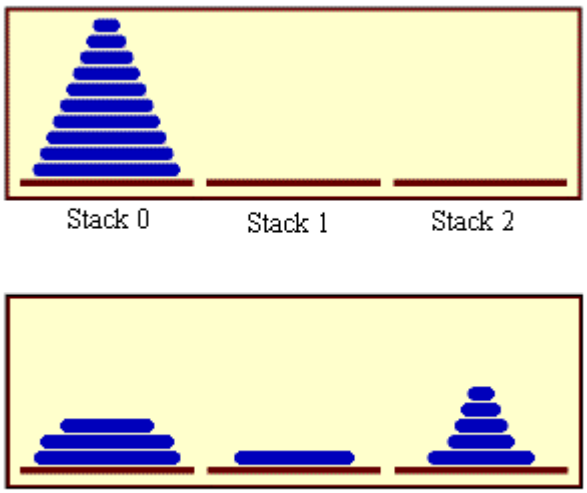
Most people find it difficult at first to convince themselves that recursion actually works. The key is to note two things that must be true for recursion to work properly: There must be one or more base cases, which can be handled without using recursion. And when recursion is applied during the solution of a problem, it must be applied to a problem that is in some sense smaller—that is, closer to the base cases—than the original problem. The idea is that if you can solve small problems and if you can reduce big problems to smaller problems, then you can solve problems of any size. Ultimately, of course, the big problems have to be reduced, possibly in many, many steps, to the very smallest problems (the base cases). Doing so might involve an immense amount of detailed bookkeeping. But the computer does that bookkeeping, not you! As a programmer, you lay out the big picture: the base cases and the reduction of big problems to smaller problems. The computer takes care of the details involved in reducing a big problem, in many steps, all the way down to base cases. Trying to think through this reduction in detail is likely to drive you crazy, and will probably make you think that recursion is hard. Whereas in fact, recursion is an elegant and powerful method that is often the simplest approach to solving a complex problem.

A common error in writing recursive subroutines is to violate one of the two rules: There must be one or more base cases, and when the subroutine is applied recursively, it must be applied to a problem that is smaller than the original problem. If these rules are violated, the

result can be an ***infinite recursion***, where the subroutine keeps calling itself over and over, without ever reaching a base case. Infinite recursion is similar to an infinite loop. However, since each recursive call to the subroutine uses up some of the computer's memory, a program that is stuck in an infinite recursion will run out of memory and crash before long. In Java, the program will crash with an exception of type `StackOverflowError`.

9.1.2 Towers of Hanoi

We have been studying an algorithm, binary search, that can easily be implemented with a `while` loop, instead of with recursion. Next, we turn to a problem that is easy to solve with recursion but difficult to solve without it. This is a standard example known as "The Towers of Hanoi." The problem involves a stack of various-sized disks, piled up on a base in order of decreasing size. The object is to move the stack from one base to another, subject to two rules: Only one disk can be moved at a time, and no disk can ever be placed on top of a smaller disk. There is a third base that can be used as a "spare." The starting situation for a stack of ten disks is shown in the top half of the following picture. The situation after a number of moves have been made is shown in the bottom half of the picture. These pictures are from the applet at the end of Section 9.5 in the on-line version of this book, which displays an animation of the step-by-step solution of the problem.



The stacks after a number of moves.

The problem is to move ten disks from Stack 0 to Stack 1, subject to certain rules. Stack 2 can be used as a spare location. Can we reduce this to smaller problems of the same type, possibly generalizing the problem a bit to make this possible? It seems natural to consider the size of the problem to be the number of disks to be moved. If there are `N` disks in Stack 0, we know that we will eventually have to move the bottom disk from Stack 0 to Stack 1. But before we can do that, according to the rules, the first `N-1` disks must be on Stack 2. Once we've moved the `N`-th disk to Stack 1, we must move the other `N-1` disks from Stack 2 to Stack 1 to complete the solution. But moving `N-1` disks is the same type of problem as moving `N` disks, except that it's a smaller version of the problem. This is exactly what we need to do recursion! The problem has to be generalized a bit, because the smaller problems involve moving disks from Stack 0 to Stack 2 or from Stack 2 to Stack 1, instead of from Stack 0 to Stack 1. In the

recursive subroutine that solves the problem, the stacks that serve as the source and destination of the disks have to be specified. It's also convenient to specify the stack that is to be used as a spare, even though we could figure that out from the other two parameters. The base case is when there is only one disk to be moved. The solution in this case is trivial: Just move the disk in one step. Here is a version of the subroutine that will print out step-by-step instructions for solving the problem:

```
/**
 * Solve the problem of moving the number of disks specified
 * by the first parameter, from the stack specified by the
 * second parameter, to the stack specified by the third
 * parameter.  The stack specified by the fourth parameter
 * is available for use as a spare.  Stacks are specified by
 * number: 0, 1, or 2.  Precondition: The number of disks is
 * a positive number.
 */
static void TowersOfHanoi(int disks, int from, int to, int spare) {
   if (disks == 1) {
         // There is only one disk to be moved.  Just move it.
      System.out.println("Move a disk from stack number "
               + from + " to stack number " + to);
   }
   else {
         // Move all but one disk to the spare stack, then
         // move the bottom disk, then put all the other
         // disks on top of it.
      TowersOfHanoi(disks-1, from, spare, to);
      System.out.println("Move a disk from stack number "
               + from + " to stack number " + to);
      TowersOfHanoi(disks-1, spare, to, from);
   }
}
```

This subroutine just expresses the natural recursive solution. The recursion works because each recursive call involves a smaller number of disks, and the problem is trivial to solve in the base case, when there is only one disk. To solve the "top level" problem of moving `N` disks from Stack 0 to Stack 1, it should be called with the command `TowersOfHanoi(N,0,1,2)`. The subroutine is demonstrated by the sample program *TowersOfHanoi.java*.

Here, for example, is the output from the program when it is run with the number of disks set equal to 4:

```
Move a disk from stack number 0 to stack number 2
Move a disk from stack number 0 to stack number 1
Move a disk from stack number 2 to stack number 1
Move a disk from stack number 0 to stack number 2
Move a disk from stack number 1 to stack number 0
Move a disk from stack number 1 to stack number 2
Move a disk from stack number 0 to stack number 2
Move a disk from stack number 0 to stack number 1
Move a disk from stack number 2 to stack number 1
Move a disk from stack number 2 to stack number 0
Move a disk from stack number 1 to stack number 0
Move a disk from stack number 2 to stack number 1
```

```
Move a disk from stack number 0 to stack number 2
Move a disk from stack number 0 to stack number 1
Move a disk from stack number 2 to stack number 1
```

The output of this program shows you a mass of detail that you don't really want to think about! The difficulty of following the details contrasts sharply with the simplicity and elegance of the recursive solution. Of course, you really want to leave the details to the computer. It's much more interesting to watch the applet from Section 9.5, which shows the solution graphically. That applet uses the same recursive subroutine, except that the `System.out.println` statements are replaced by commands that show the image of the disk being moved from one stack to another. (You might think about what happens when the precondition that the number of disks is positive is violated. The result is an example of infinite recursion.)

There is, by the way, a story that explains the name of this problem. According to this story, on the first day of creation, a group of monks in an isolated tower near Hanoi were given a stack of 64 disks and were assigned the task of moving one disk every day, according to the rules of the Towers of Hanoi problem. On the day that they complete their task of moving all the disks from one stack to another, the universe will come to an end. But don't worry. The number of steps required to solve the problem for `N` disks is 2^N - 1, and 2^{64} - 1 days is over 50,000,000,000,000 years. We have a long way to go.

(In the terminology of Section 8.5, the Towers of Hanoi algorithm has a run time that is $\Theta(2^n)$, where n is the number of disks that have to be moved. Since the exponential function 2^n grows so quickly, the Towers of Hanoi problem can be solved in practice only for a small number of disks.)

* * *

By the way, in addition to the graphical Towers of Hanoi applet at the end of this chapter, there are three other end-of-chapter applets in the on-line version of this text that use recursion. One, at the end of Section 12.5, is a visual implementation of the Quicksort algorithm that is discussed below. One is a maze-solving applet, at the end of Section 11.5. And the other is a pentominos applet, at the end of Section 10.5.

The Maze applet first builds a random maze. It then tries to solve the maze by finding a path through the maze from the upper left corner to the lower right corner. This problem is actually very similar to a "blob-counting" problem that is considered later in this section. The recursive maze-solving routine starts from a given square, and it visits each neighboring square and calls itself recursively from there. The recursion ends if the routine finds itself at the lower right corner of the maze.

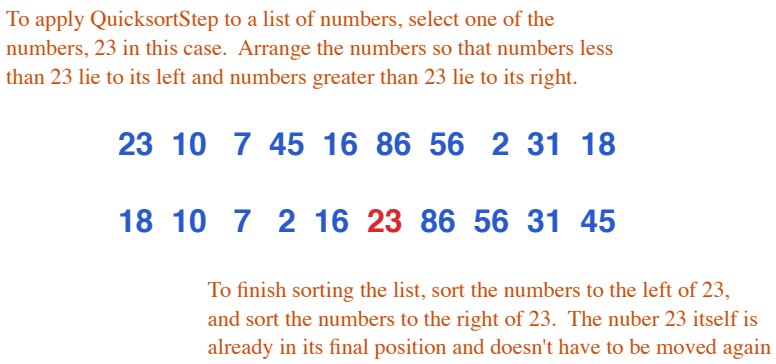
The Pentominos applet is an implementation of a classic puzzle. A pentomino is a connected figure made up of five equal-sized squares. There are exactly twelve figures that can be made in this way, not counting all the possible rotations and reflections of the basic figures. The problem is to place the twelve pentominos on an 8-by-8 board in which four of the squares have already been marked as filled. The recursive solution looks at a board that has already been partially filled with pentominos. The subroutine looks at each remaining piece in turn. It tries to place that piece in the next available place on the board. If the piece fits, it calls itself recursively to try to fill in the rest of the solution. If that fails, then the subroutine goes on to the next piece. A generalized version of the pentominos applet with many more features can be found at http://math.hws.edu/xJava/PentominosSolver/.

The applets are fun to watch, and they give nice visual representations of recursion.

9.1.3 A Recursive Sorting Algorithm

Turning next to an application that is perhaps more practical, we'll look at a recursive algorithm for sorting an array. The selection sort and insertion sort algorithms, which were covered in Section 7.4, are fairly simple, but they are rather slow when applied to large arrays. Faster sorting algorithms are available. One of these is ***Quicksort***, a recursive algorithm which turns out to be the fastest sorting algorithm in most situations.

The Quicksort algorithm is based on a simple but clever idea: Given a list of items, select any item from the list. This item is called the ***pivot***. (In practice, I'll just use the first item in the list.) Move all the items that are smaller than the pivot to the beginning of the list, and move all the items that are larger than the pivot to the end of the list. Now, put the pivot between the two groups of items. This puts the pivot in the position that it will occupy in the final, completely sorted array. It will not have to be moved again. We'll refer to this procedure as QuicksortStep.



QuicksortStep is not recursive. It is used as a subroutine by Quicksort. The speed of Quicksort depends on having a fast implementation of QuicksortStep. Since it's not the main point of this discussion, I present one without much comment.

```
/**
 * Apply QuicksortStep to the list of items in locations lo through hi
 * in the array A.  The value returned by this routine is the final
 * position of the pivot item in the array.
 */
static int quicksortStep(int[] A, int lo, int hi) {

   int pivot = A[lo];  // Get the pivot value.

   // The numbers hi and lo mark the endpoints of a range
   // of numbers that have not yet been tested.  Decrease hi
   // and increase lo until they become equal, moving numbers
   // bigger than pivot so that they lie above hi and moving
   // numbers less than the pivot so that they lie below lo.
   // When we begin, A[lo] is an available space, since its
   // value has been moved into the local variable, pivot.

   while (hi > lo) {

      while (hi > lo && A[hi] >= pivot) {
            // Move hi down past numbers greater than pivot.
            // These numbers do not have to be moved.
```

```
            hi--;
         }

         if (hi == lo)
            break;

         // The number A[hi] is less than pivot.  Move it into
         // the available space at A[lo], leaving an available
         // space at A[hi].

         A[lo] = A[hi];
         lo++;

         while (hi > lo && A[lo] <= pivot) {
               // Move lo up past numbers less than pivot.
               // These numbers do not have to be moved.
            lo++;
         }

         if (hi == lo)
            break;

         // The number A[lo] is greater than pivot.  Move it into
         // the available space at A[hi], leaving an available
         // space at A[lo].

         A[hi] = A[lo];
         hi--;

      } // end while

      // At this point, lo has become equal to hi, and there is
      // an available space at that position.  This position lies
      // between numbers less than pivot and numbers greater than
      // pivot.  Put pivot in this space and return its location.

      A[lo] = pivot;
      return lo;

   }  // end QuicksortStep
```

With this subroutine in hand, Quicksort is easy. The Quicksort algorithm for sorting a list consists of applying QuicksortStep to the list, then applying Quicksort recursively to the items that lie to the left of the new position of the pivot and to the items that lie to the right of that position. Of course, we need base cases. If the list has only one item, or no items, then the list is already as sorted as it can ever be, so Quicksort doesn't have to do anything in these cases.

```
/**
 * Apply quicksort to put the array elements between
 * position lo and position hi into increasing order.
 */
static void quicksort(int[] A, int lo, int hi) {
   if (hi <= lo) {
         // The list has length one or zero.  Nothing needs
         // to be done, so just return from the subroutine.
      return;
   }
   else {
         // Apply quicksortStep and get the new pivot position.
```

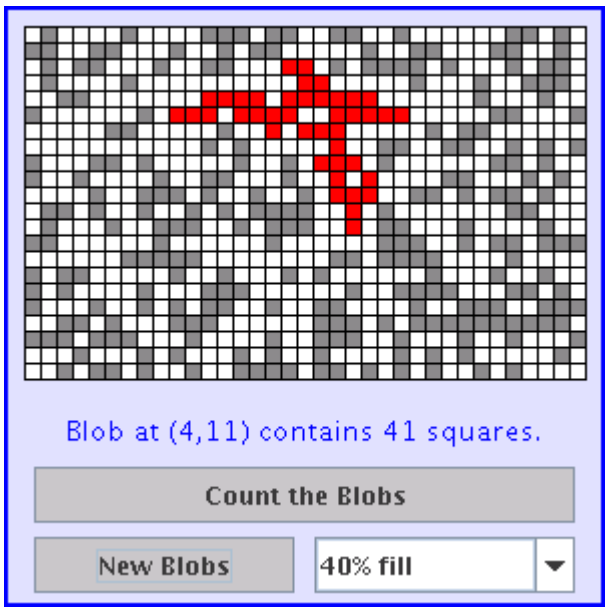
```
            // Then apply quicksort to sort the items that
            // precede the pivot and the items that follow it.
         int pivotPosition = quicksortStep(A, lo, hi);
         quicksort(A, lo, pivotPosition - 1);
         quicksort(A, pivotPosition + 1, hi);
      }
   }
```

As usual, we had to generalize the problem. The original problem was to sort an array, but the recursive algorithm is set up to sort a specified part of an array. To sort an entire array, `A`, using the `quickSort()` subroutine, you would call `quicksort(A, 0, A.length - 1)`.

Quicksort is an interesting example from the point of view of the analysis of algorithms (Section 8.5), because its average case run time differs greatly from its worst case run time. Here is a very informal analysis, starting with the average case: Note that an application of quicksortStep divides a problem into two sub-problems. On the average, the subproblems will be of approximately the same size. A problem of size n is divided into two problems that are roughly of size n/2; these are then divided into four problems that are roughly of size n/4; and so on. Since the problem size is divided by 2 on each level, there will be approximately log(n) levels of subdivision. The amount of processing on each level is proportional to n. (On the top level, each element in the array is looked at and possibly moved. On the second level, where there are two subproblems, every element but one in the array is part of one of those two subproblems and must be looked at and possibly moved, so there is a total of about n steps in both subproblems combined. Similarly, on the third level, there are four subproblems and a total of about n steps in the four subproblems on that level....) With a total of n steps on each level and approximately log(n) levels in the average case, the average case run time for Quicksort is Θ(n*log(n)). This analysis assumes that quicksortStep divides a problem into two approximately equal parts. However, in the worst case, each application of quicksortStep divides a problem of size n into a problem of size 0 and a problem of size n-1. This happens when the pivot element ends up at the beginning or end of the array. In this worst case, there are n levels of subproblems, and the worst-case run time is $\Theta(n^2)$. The worst case is very rare—it depends on the items in the array being arranged in a very special way, so the average performance of Quicksort can be very good even though it is not so good in certain rare cases. There are sorting algorithms that have both an average case and a worst case run time of Θ(n*log(n)). One example is MergeSort, which you can look up if you are interested.

9.1.4 Blob Counting

The program *Blobs.java* displays a grid of small white and gray squares. The gray squares are considered to be "filled" and the white squares are "empty." For the purposes of this example, we define a "blob" to consist of a filled square and all the filled squares that can be reached from it by moving up, down, left, and right through other filled squares. If the user clicks on any filled square in the program, the computer will count the squares in the blob that contains the clicked square, and it will change the color of those squares to red. The program has several controls. There is a "New Blobs" button; clicking this button will create a new random pattern in the grid. A pop-up menu specifies the approximate percentage of squares that will be filled in the new pattern. The more filled squares, the larger the blobs. And a button labeled "Count the Blobs" will tell you how many different blobs there are in the pattern. You can try an applet version of the program in the on-line version of the book. Here is a picture of the program after the user has clicked one of the filled squares:



Recursion is used in this program to count the number of squares in a blob. Without recursion, this would be a very difficult thing to implement. Recursion makes it relatively easy, but it still requires a new technique, which is also useful in a number of other applications.

The data for the grid of squares is stored in a two dimensional array of boolean values,

```
boolean[][]  filled;
```

The value of `filled[r][c]` is true if the square in row `r` and in column `c` of the grid is filled. The number of rows in the grid is stored in an instance variable named `rows`, and the number of columns is stored in `columns`. The program uses a recursive instance method named `getBlobSize()` to count the number of squares in the blob that contains the square in a given row `r` and column `c`. If there is no filled square at position `(r,c)`, then the answer is zero. Otherwise, `getBlobSize()` has to count all the filled squares that can be reached from the square at position `(r,c)`. The idea is to use `getBlobSize()` recursively to get the number of filled squares that can be reached from each of the neighboring positions: `(r+1,c)`, `(r-1,c)`, `(r,c+1)`, and `(r,c-1)`. Add up these numbers, and add one to count the square at `(r,c)` itself, and you get the total number of filled squares that can be reached from `(r,c)`. Here is an implementation of this algorithm, as stated. Unfortunately, it has a serious flaw: It leads to an infinite recursion!

```
int getBlobSize(int r, int c) {  // BUGGY, INCORRECT VERSION!!
      // This INCORRECT method tries to count all the filled
      // squares that can be reached from position (r,c) in the grid.
   if (r < 0 || r >= rows || c < 0 || c >= columns) {
         // This position is not in the grid, so there is
         // no blob at this position.  Return a blob size of zero.
      return 0;
   }
   if (filled[r][c] == false) {
        // This square is not part of a blob, so return zero.
      return 0;
   }
   int size = 1;  // Count the square at this position, then count the
```

```
                //   the blobs that are connected to this square
                //   horizontally or vertically.
   size += getBlobSize(r-1,c);
   size += getBlobSize(r+1,c);
   size += getBlobSize(r,c-1);
   size += getBlobSize(r,c+1);
   return size;
}  // end INCORRECT getBlobSize()
```

Unfortunately, this routine will count the same square more than once. In fact, it will try to count each square infinitely often! Think of yourself standing at position `(r,c)` and trying to follow these instructions. The first instruction tells you to move up one row. You do that, and then you apply the same procedure. As one of the steps in that procedure, you have to move **down** one row and apply the same procedure yet again. But that puts you back at position `(r,c)`! From there, you move up one row, and from there you move down one row.... Back and forth forever! We have to make sure that a square is only counted and processed once, so we don't end up going around in circles. The solution is to leave a trail of breadcrumbs—or on the computer a trail of boolean values—to mark the squares that you've already visited. Once a square is marked as visited, it won't be processed again. The remaining, unvisited squares are reduced in number, so definite progress has been made in reducing the size of the problem. Infinite recursion is avoided!

A second boolean array, `visited[r][c]`, is used to keep track of which squares have already been visited and processed. It is assumed that all the values in this array are set to false before `getBlobSize()` is called. As `getBlobSize()` encounters unvisited squares, it marks them as visited by setting the corresponding entry in the `visited` array to `true`. When `getBlobSize()` encounters a square that it has already visited, it doesn't count it or process it further. The technique of "marking" items as they are encountered is one that is used over and over in the programming of recursive algorithms. Here is the corrected version of `getBlobSize()`, with changes shown in italic:

```
/**
 * Counts the squares in the blob at position (r,c) in the
 * grid.  Squares are only counted if they are filled and
 * unvisited.  If this routine is called for a position that
 * has been visited, the return value will be zero.
 */
int getBlobSize(int r, int c) {
   if (r < 0 || r >= rows || c < 0 || c >= columns) {
         // This position is not in the grid, so there is
         // no blob at this position.  Return a blob size of zero.
      return 0;
   }
   if (filled[r][c] == false || visited[r][c] == true) {
        // This square is not part of a blob, or else it has
        // already been counted, so return zero.
      return 0;
   }
   visited[r][c] = true;   // Mark the square as visited so that
                           //    we won't count it again during the
                           //    following recursive calls.
   int size = 1;  // Count the square at this position, then count the
                  //   the blobs that are connected to this square
```

```
                //   horizontally or vertically.
   size += getBlobSize(r-1,c);
   size += getBlobSize(r+1,c);
   size += getBlobSize(r,c-1);
   size += getBlobSize(r,c+1);
   return size;
}  // end getBlobSize()
```

In the program, this method is used to determine the size of a blob when the user clicks on a square. After `getBlobSize()` has performed its task, all the squares in the blob are still marked as visited. The `paintComponent()` method draws visited squares in red, which makes the blob visible. The `getBlobSize()` method is also used for counting blobs. This is done by the following method, which includes comments to explain how it works:

```
/**
 * When the user clicks the "Count the Blobs" button, find the
 * number of blobs in the grid and report the number in the
 * message label.
 */
void countBlobs() {

   int count = 0; // Number of blobs.

   /* First clear out the visited array. The getBlobSize() method
      will mark every filled square that it finds by setting the
      corresponding element of the array to true.  Once a square
      has been marked as visited, it will stay marked until all the
      blobs have been counted.  This will prevent the same blob from
      being counted more than once. */

   for (int r = 0; r < rows; r++)
      for (int c = 0; c < columns; c++)
         visited[r][c] = false;

   /* For each position in the grid, call getBlobSize() to get the
      size of the blob at that position.  If the size is not zero,
      count a blob.  Note that if we come to a position that was part
      of a previously counted blob, getBlobSize() will return 0 and
      the blob will not be counted again. */

   for (int r = 0; r < rows; r++)
      for (int c = 0; c < columns; c++) {
         if (getBlobSize(r,c) > 0)
            count++;
      }

   repaint();  // Note that all the filled squares will be red,
               //   since they have all now been visited.

   message.setText("The number of blobs is " + count);

} // end countBlobs()
```

9.2 Linked Data Structures

EVERY USEFUL OBJECT contains instance variables. When the type of an instance variable is given by a class or interface name, the variable can hold a reference to another object. Such a reference is also called a pointer, and we say that the variable ***points to*** the object. (Of course, any variable that can contain a reference to an object can also contain the special value `null`, which points to nowhere.) When one object contains an instance variable that points to another object, we think of the objects as being "linked" by the pointer. Data structures of great complexity can be constructed by linking objects together.

9.2.1 Recursive Linking

Something interesting happens when an object contains an instance variable that can refer to another object of the same type. In that case, the definition of the object's class is recursive. Such recursion arises naturally in many cases. For example, consider a class designed to represent employees at a company. Suppose that every employee except the boss has a supervisor, who is another employee of the company. Then the *Employee* class would naturally contain an instance variable of type *Employee* that points to the employee's supervisor:

```
/**
 * An object of type Employee holds data about one employee.
 */
public class Employee {

   String name;          // Name of the employee.

   Employee supervisor;  // The employee's supervisor.

      .
      .  // (Other instance variables and methods.)
      .

} // end class Employee
```

If `emp` is a variable of type *Employee*, then `emp.supervisor` is another variable of type *Employee*. If `emp` refers to the boss, then the value of `emp.supervisor` should be `null` to indicate the fact that the boss has no supervisor. If we wanted to print out the name of the employee's supervisor, for example, we could use the following Java statement:

```
if ( emp.supervisor == null) {
   System.out.println( emp.name + " is the boss and has no supervisor!" );
}
else {
   System.out.print( "The supervisor of " + emp.name + " is " );
   System.out.println( emp.supervisor.name );
}
```

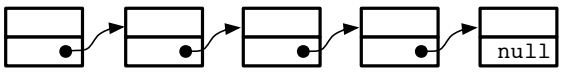
Now, suppose that we want to know how many levels of supervisors there are between a given employee and the boss. We just have to follow the chain of command through a series of `supervisor` links, and count how many steps it takes to get to the boss:

```
if ( emp.supervisor == null ) {
   System.out.println( emp.name + " is the boss!" );
}
else {
```

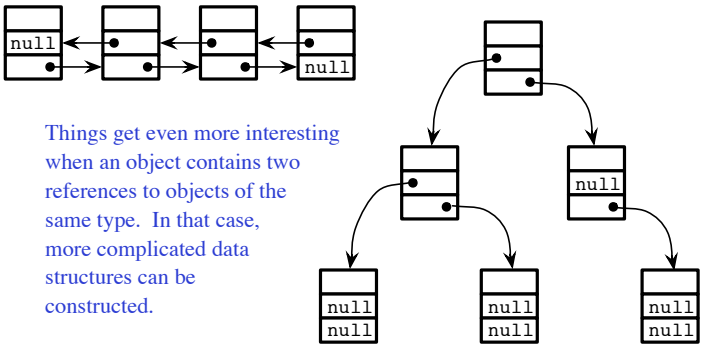
```
        Employee runner;  // For "running" up the chain of command.
        runner = emp.supervisor;
        if ( runner.supervisor == null) {
           System.out.println( emp.name  + " reports directly to the boss." );
        }
        else {
           int count = 0;
           while ( runner.supervisor != null ) {
              count++;  // Count the supervisor on this level.
              runner = runner.supervisor; // Move up to the next level.
           }
           System.out.println( "There are " + count
                                  + " supervisors between " + emp.name
                                  + " and the boss." );
        }
     }
```

As the `while` loop is executed, `runner` points in turn to the original employee (`emp`), then to `emp's` supervisor, then to the supervisor of `emp's` supervisor, and so on. The `count` variable is incremented each time `runner` "visits" a new employee. The loop ends when `runner.supervisor` is `null`, which indicates that `runner` has reached the boss. At that point, `count` has counted the number of steps between `emp` and the boss.

In this example, the `supervisor` variable is quite natural and useful. In fact, data structures that are built by linking objects together are so useful that they are a major topic of study in computer science. We'll be looking at a few typical examples. In this section and the next, we'll be looking at ***linked lists***. A linked list consists of a chain of objects of the same type, linked together by pointers from one object to the next. This is much like the chain of supervisors between `emp` and the boss in the above example. It's also possible to have more complex situations, in which one object can contain links to several other objects. We'll look at an example of this in Section 9.4.



When an object contains a reference to an object of the same type, then several objects can be linked together into a list. Each object refers to the next object.



Things get even more interesting when an object contains two references to objects of the same type. In that case, more complicated data structures can be constructed.

9.2.2 Linked Lists

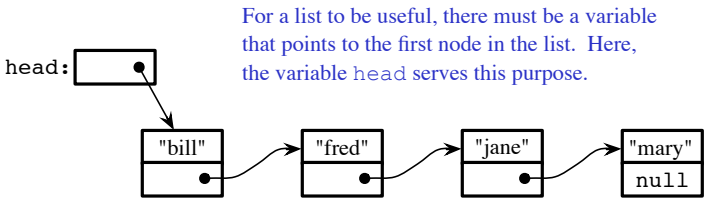
For most of the examples in the rest of this section, linked lists will be constructed out of objects belonging to the class *Node* which is defined as follows:

```
class Node {
   String item;
   Node next;
}
```

The term ***node*** is often used to refer to one of the objects in a linked data structure. Objects of type *Node* can be chained together as shown in the top part of the above picture. Each node holds a *String* and a pointer to the next node in the list (if any). The last node in such a list can always be identified by the fact that the instance variable `next` in the last node holds the value `null` instead of a pointer to another node. The purpose of the chain of nodes is to represent a list of strings. The first string in the list is stored in the first node, the second string is stored in the second node, and so on. The pointers and the node objects are used to build the structure, but the data that we want to represent is the list of strings. Of course, we could just as easily represent a list of integers or a list of *JButtons* or a list of any other type of data by changing the type of the `item` that is stored in each node.

Although the *Nodes* in this example are very simple, we can use them to illustrate the common operations on linked lists. Typical operations include deleting nodes from the list, inserting new nodes into the list, and searching for a specified *String* among the `items` in the list. We will look at subroutines to perform all of these operations, among others.

For a linked list to be used in a program, that program needs a variable that refers to the first node in the list. It only needs a pointer to the first node since all the other nodes in the list can be accessed by starting at the first node and following links along the list from one node to the next. In my examples, I will always use a variable named `head`, of type *Node*, that points to the first node in the linked list. When the list is empty, the value of `head` is null.



9.2.3 Basic Linked List Processing

It is very common to want to process all the items in a linked list in some way. The common pattern is to start at the head of the list, then move from each node to the next by following the pointer in the node, stopping when the `null` that marks the end of the list is reached. If `head` is a variable of type *Node* that points to the first node in the list, then the general form of the code for processing all the items in a linked list is:

```
Node runner;    // A pointer that will be used to traverse the list.
runner = head;  // Start with runner pointing to the head of the list.
while ( runner != null ) {     // Continue until null is encountered.
   process( runner.item );     // Do something with the item in the current node.
```

```
   runner = runner.next;       // Move on to the next node in the list.
}
```

Our only access to the list is through the variable `head`, so we start by getting a copy of the value in `head` with the assignment statement `runner = head`. We need a **copy** of `head` because we are going to change the value of `runner`. We can't change the value of `head`, or we would lose our only access to the list! The variable `runner` will point to each node of the list in turn. When `runner` points to one of the nodes in the list, `runner.next` is a pointer to the next node in the list, so the assignment statement `runner = runner.next` moves the pointer along the list from each node to the next. We know that we've reached the end of the list when `runner` becomes equal to `null`. Note that our list-processing code works even for an empty list, since for an empty list the value of `head` is `null` and the body of the while loop is not executed at all. As an example, we can print all the strings in a list of *Strings* by saying:

```
Node runner = head;
while ( runner != null ) {
   System.out.println(  runner.item );
   runner = runner.next;
}
```

The `while` loop can, by the way, be rewritten as a `for` loop. Remember that even though the loop control variable in a `for` loop is often numerical, that is not a requirement. Here is a `for` loop that is equivalent to the above `while` loop:

```
for ( Node runner = head; runner != null; runner = runner.next ) {
   System.out.println( runner.item );
}
```

Similarly, we can traverse a list of integers to add up all the numbers in the list. A linked list of integers can be constructed using the class

```
public class IntNode {
   int item;       // One of the integers in the list.
   IntNode next;   // Pointer to the next node in the list.
}
```

If `head` is a variable of type *IntNode* that points to a linked list of integers, we can find the sum of the integers in the list using:

```
int sum = 0;
IntNode runner = head;
while ( runner != null ) {
   sum = sum + runner.item;   // Add current item to the sum.
   runner = runner.next;
}
System.out.println("The sum of the list of items is " + sum);
```

It is also possible to use recursion to process a linked list. Recursion is rarely the natural way to process a list, since it's so easy to use a loop to traverse the list. However, understanding how to apply recursion to lists can help with understanding the recursive processing of more complex data structures. A non-empty linked list can be thought of as consisting of two parts: the ***head*** of the list, which is just the first node in the list, and the ***tail*** of the list, which consists of the remainder of the list after the head. Note that the tail is itself a linked list and that it is shorter than the original list (by one node). This is a natural setup for recursion, where the problem of processing a list can be divided into processing the head and recursively

processing the tail. The base case occurs in the case of an empty list (or sometimes in the case of a list of length one). For example, here is a recursive algorithm for adding up the numbers in a linked list of integers:

```
if the list is empty then
   return 0 (since there are no numbers to be added up)
otherwise
   let listsum = the number in the head node
   let tailsum be the sum of the numbers in the tail list (recursively)
   add tailsum to listsum
   return listsum
```

One remaining question is, how do we get the tail of a non-empty linked list? If `head` is a variable that points to the head node of the list, then `head.next` is a variable that points to the second node of the list—and that node is in fact the first node of the tail. So, we can view `head.next` as a pointer to the tail of the list. One special case is when the original list consists of a single node. In that case, the tail of the list is empty, and `head.next` is `null`. Since an empty list is represented by a null pointer, `head.next` represents the tail of the list even in this special case. This allows us to write a recursive list-summing function in Java as

```
/**
 *  Compute the sum of all the integers in a linked list of integers.
 *  @param head a pointer to the first node in the linked list
 */
public static int addItemsInList( IntNode head ) {
   if ( head == null ) {
         // Base case:  The list is empty,  so the sum is zero.
      return 0;
   }
   else {
         // Recursive case:  The list is non-empty.  Find the sum of
         // the tail list, and add that to the item in the head node.
         // (Note that this case could be written simply as
         //     return head.item + addItemsInList( head.next );)
      int listsum = head.item;
      int tailsum = addItemsInList( head.next );
      listsum = listsum + tailsum;
      return listsum;
   }
}
```

I will finish by presenting a list-processing problem that is easy to solve with recursion, but quite tricky to solve without it. The problem is to print out all the strings in a linked list of strings in the **reverse** of the order in which they occur in the list. Note that when we do this, the item in the head of a list is printed out after all the items in the tail of the list. This leads to the following recursive routine. You should convince yourself that it works, and you should think about trying to do the same thing without using recursion:

```
public static void printReversed( Node head ) {
   if ( head == null ) {
         // Base case:  The list is empty, and there is nothing to print.
      return;
   }
   else {
```

```
            // Recursive case:  The list is non-empty.
         printReversed( head.next );  // Print strings from tail, in reverse order.
         System.out.println( head.item );  // Then print string from head node.
      }
   }
```

* * *

In the rest of this section, we'll look at a few more advanced operations on a linked list of strings. The subroutines that we consider are instance methods in a class, *StringList*. An object of type *StringList* represents a linked list of strings. The class has a private instance variable named `head` of type *Node* that points to the first node in the list, or is null if the list is empty. Instance methods in class *StringList* access `head` as a global variable. The source code for *StringList* is in the file *StringList.java*, and it is used in the sample program *ListDemo.java*.

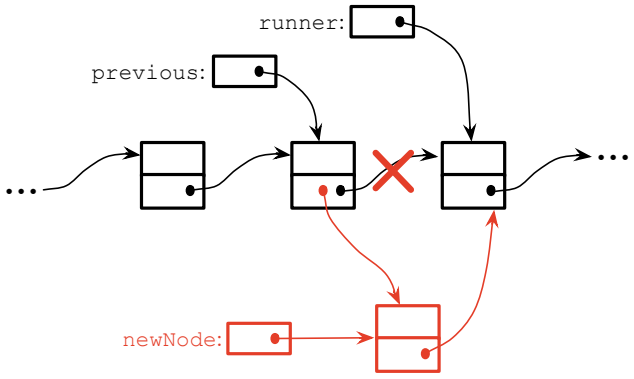
Suppose we want to know whether a specified string, `searchItem`, occurs somewhere in a list of strings. We have to compare `searchItem` to each `item` in the list. This is an example of basic list traversal and processing. However, in this case, we can stop processing if we find the item that we are looking for.

```
/**
 * Searches the list for a specified item.
 * @param searchItem the item that is to be searched for
 * @return true if searchItem is one of the items in the list or false if
 *    searchItem does not occur in the list.
 */
public boolean find(String searchItem) {

   Node runner;    // A pointer for traversing the list.

   runner = head;  // Start by looking at the head of the list.
                   //   (head is an instance variable! )

   while ( runner != null ) {
         // Go through the list looking at the string in each
         // node.  If the string is the one we are looking for,
         // return true, since the string has been found in the list.
      if ( runner.item.equals(searchItem) )
         return true;
      runner = runner.next;  // Move on to the next node.
   }

   // At this point, we have looked at all the items in the list
   // without finding searchItem.  Return false to indicate that
   // the item does not exist in the list.

   return false;

} // end find()
```

It is possible that the list is empty, that is, that the value of `head` is `null`. We should be careful that this case is handled properly. In the above code, if `head` is `null`, then the body of the `while` loop is never executed at all, so no nodes are processed and the return value is `false`. This is exactly what we want when the list is empty, since the `searchItem` can't occur in an empty list.

9.2.4 Inserting into a Linked List

The problem of inserting a new item into a linked list is more difficult, at least in the case where the item is inserted into the middle of the list. (In fact, it's probably the most difficult operation on linked data structures that you'll encounter in this chapter.) In the *StringList* class, the `items` in the nodes of the linked list are kept in increasing order. When a new item is inserted into the list, it must be inserted at the correct position according to this ordering. This means that, usually, we will have to insert the new item somewhere in the middle of the list, between two existing nodes. To do this, it's convenient to have two variables of type *Node*, which refer to the existing nodes that will lie on either side of the new node. In the following illustration, these variables are `previous` and `runner`. Another variable, `newNode`, refers to the new node. In order to do the insertion, the link from `previous` to `runner` must be "broken," and new links from `previous` to `newNode` and from `newNode` to `runner` must be added:



Inserting a new node
into the middle of a list

Once we have `previous` and `runner` pointing to the right nodes, the command "`previous.next = newNode;`" can be used to make `previous.next` point to the new node, instead of to the node indicated by `runner`. And the command "`newNode.next = runner`" will set `newNode.next` to point to the correct place. However, before we can use these commands, we need to set up `runner` and `previous` as shown in the illustration. The idea is to start at the first node of the list, and then move along the list past all the items that are less than the new item. While doing this, we have to be aware of the danger of "falling off the end of the list." That is, we can't continue if `runner` reaches the end of the list and becomes `null`. If `insertItem` is the item that is to be inserted, and if we assume that it does, in fact, belong somewhere in the middle of the list, then the following code would correctly position `previous` and `runner`:

```
Node runner, previous;
previous = head;     // Start at the beginning of the list.
runner = head.next;
while ( runner != null && runner.item.compareTo(insertItem) < 0 ) {
   previous = runner;  // "previous = previous.next" would also work
   runner = runner.next;
}
```

(This uses the `compareTo()` instance method from the *String* class to test whether the item in the node is less than the item that is being inserted. See Subsection 2.3.2.)

This is fine, except that the assumption that the new node is inserted into the middle of the list is not always valid. It might be that `insertItem` is less than the first item of the list. In that case, the new node must be inserted at the head of the list. This can be done with the instructions

```
newNode.next = head;   // Make newNode.next point to the old head.
head = newNode;        // Make newNode the new head of the list.
```

It is also possible that the list is empty. In that case, `newNode` will become the first and only node in the list. This can be accomplished simply by setting `head = newNode`. The following `insert()` method from the *StringList* class covers all of these possibilities:

```
/**
 * Insert a specified item to the list, keeping the list in order.
 * @param insertItem the item that is to be inserted.
 */
public void insert(String insertItem) {

   Node newNode;          // A Node to contain the new item.
   newNode = new Node();
   newNode.item = insertItem;  // (N.B.  newNode.next is null.)

   if ( head == null ) {
          // The new item is the first (and only) one in the list.
          // Set head to point to it.
      head = newNode;
   }
   else if ( head.item.compareTo(insertItem) >= 0 ) {
          // The new item is less than the first item in the list,
          // so it has to be inserted at the head of the list.
      newNode.next = head;
      head = newNode;
   }
   else {
          // The new item belongs somewhere after the first item
          // in the list.  Search for its proper position and insert it.
      Node runner;     // A node for traversing the list.
      Node previous;   // Always points to the node preceding runner.
      runner = head.next;   // Start by looking at the SECOND position.
      previous = head;
      while ( runner != null && runner.item.compareTo(insertItem) < 0 ) {
             // Move previous and runner along the list until runner
             // falls off the end or hits a list element that is
             // greater than or equal to insertItem.  When this
             // loop ends, previous indicates the position where
             // insertItem must be inserted.
         previous = runner;
         runner = runner.next;
      }
      newNode.next = runner;     // Insert newNode after previous.
      previous.next = newNode;
   }

}  // end insert()
```

If you were paying close attention to the above discussion, you might have noticed that there is one special case which is not mentioned. What happens if the new node has to be inserted at the **end** of the list? This will happen if all the items in the list are less than the new item. In fact, this case is already handled correctly by the subroutine, in the last part of the `if` statement. If `insertItem` is greater than all the items in the list, then the `while` loop will end when `runner` has traversed the entire list and become `null`. However, when that happens, `previous` will be left pointing to the last node in the list. Setting `previous.next = newNode` adds `newNode` onto the end of the list. Since `runner` is `null`, the command `newNode.next = runner` sets `newNode.next` to `null`, which is exactly what is needed to mark the end of the list.

9.2.5 Deleting from a Linked List

The delete operation is similar to insert, although a little simpler. There are still special cases to consider. When the first node in the list is to be deleted, then the value of `head` has to be changed to point to what was previously the second node in the list. Since `head.next` refers to the second node in the list, this can be done by setting `head = head.next`. (Once again, you should check that this works when `head.next` is `null`, that is, when there is no second node in the list. In that case, the list becomes empty.)

If the node that is being deleted is in the middle of the list, then we can set up `previous` and `runner` with `runner` pointing to the node that is to be deleted and with `previous` pointing to the node that precedes that node in the list. Once that is done, the command "`previous.next = runner.next;`" will delete the node. The deleted node will be garbage collected. I encourage you to draw a picture for yourself to illustrate this operation. Here is the complete code for the `delete()` method:

```
/**
 * Delete a specified item from the list, if that item is present.
 * If multiple copies of the item are present in the list, only
 * the one that comes first in the list is deleted.
 * @param deleteItem the item to be deleted
 * @return true if the item was found and deleted, or false if the item
 *    was not in the list.
 */
public boolean delete(String deleteItem) {

   if ( head == null ) {
          // The list is empty, so it certainly doesn't contain deleteString.
      return false;
   }
   else if ( head.item.equals(deleteItem) ) {
           // The string is the first item of the list.  Remove it.
      head = head.next;
      return true;
   }
   else {
          // The string, if it occurs at all, is somewhere beyond the
          // first element of the list.  Search the list.
      Node runner;     // A node for traversing the list.
      Node previous;   // Always points to the node preceding runner.
      runner = head.next;   // Start by looking at the SECOND list node.
      previous = head;
      while ( runner != null && runner.item.compareTo(deleteItem) < 0 ) {
```

```
                // Move previous and runner along the list until runner
                // falls off the end or hits a list element that is
                // greater than or equal to deleteItem.  When this
                // loop ends, runner indicates the position where
                // deleteItem must be, if it is in the list.
             previous = runner;
             runner = runner.next;
          }
          if ( runner != null && runner.item.equals(deleteItem) ) {
                // Runner points to the node that is to be deleted.
                // Remove it by changing the pointer in the previous node.
             previous.next = runner.next;
             return true;
          }
          else {
                // The item does not exist in the list.
             return false;
          }
       }

    } // end delete()
```

9.3 Stacks, Queues, and ADTs

A LINKED LIST is a particular type of data structure, made up of objects linked together by pointers. In the previous section, we used a linked list to store an ordered list of *Strings*, and we implemented `insert`, `delete`, and `find` operations on that list. However, we could easily have stored the list of *Strings* in an array or *ArrayList*, instead of in a linked list. We could still have implemented the same operations on the list. The implementations of these operations would have been different, but their interfaces and logical behavior would still be the same.

The term ***abstract data type***, or ***ADT***, refers to a set of possible values and a set of operations on those values, without any specification of how the values are to be represented or how the operations are to be implemented. An "ordered list of strings" can be defined as an abstract data type. Any sequence of *Strings* that is arranged in increasing order is a possible value of this data type. The operations on the data type include inserting a new string, deleting a string, and finding a string in the list. There are often several different ways to implement the same abstract data type. For example, the "ordered list of strings" ADT can be implemented as a linked list or as an array. A program that only depends on the abstract definition of the ADT can use either implementation, interchangeably. In particular, the implementation of the ADT can be changed without affecting the program as a whole. This can make the program easier to debug and maintain, so ADTs are an important tool in software engineering.

In this section, we'll look at two common abstract data types, ***stacks*** and ***queues***. Both stacks and queues are often implemented as linked lists, but that is not the only possible implementation. You should think of the rest of this section partly as a discussion of stacks and queues and partly as a case study in ADTs.

9.3.1 Stacks

A stack consists of a sequence of items, which should be thought of as piled one on top of the other like a physical stack of boxes or cafeteria trays. Only the top item on the stack is

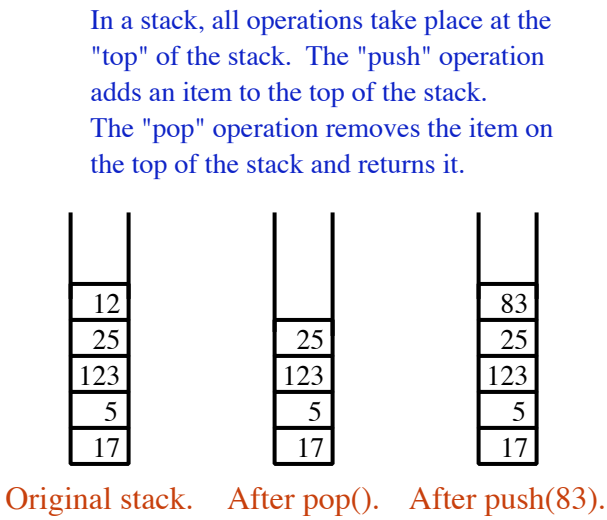
accessible at any given time. It can be removed from the stack with an operation called ***pop***. An item lower down on the stack can only be removed after all the items on top of it have been popped off the stack. A new item can be added to the top of the stack with an operation called ***push***. We can make a stack of any type of items. If, for example, the items are values of type **int**, then the push and pop operations can be implemented as instance methods

- `void push (int newItem)` — Add newItem to top of stack.
- `int pop()` — Remove the top int from the stack and return it.

It is an error to try to pop an item from an empty stack, so it is important to be able to tell whether a stack is empty. We need another stack operation to do the test, implemented as an instance method

- `boolean isEmpty()` — Returns true if the stack is empty.

This defines a "stack of ints" as an abstract data type. This ADT can be implemented in several ways, but however it is implemented, its behavior must correspond to the abstract mental image of a stack.



In the linked list implementation of a stack, the top of the stack is actually the node at the head of the list. It is easy to add and remove nodes at the front of a linked list—much easier than inserting and deleting nodes in the middle of the list. Here is a class that implements the "stack of ints" ADT using a linked list. (It uses a static nested class to represent the nodes of the linked list. If the nesting bothers you, you could replace it with a separate *Node* class.)

```
public class StackOfInts {

   /**
    * An object of type Node holds one of the items in the linked list
    * that represents the stack.
    */
   private static class Node {
      int item;
      Node next;
   }
```

```
   private Node top;  // Pointer to the Node that is at the top of
                      //   of the stack.  If top == null, then the
                      //   stack is empty.

   /**
    * Add N to the top of the stack.
    */
   public void push( int N ) {
      Node newTop;         // A Node to hold the new item.
      newTop = new Node();
      newTop.item = N;     // Store N in the new Node.
      newTop.next = top;   // The new Node points to the old top.
      top = newTop;        // The new item is now on top.
   }

   /**
    * Remove the top item from the stack, and return it.
    * Throws an IllegalStateException if the stack is empty when
    * this method is called.
    */
   public int pop() {
      if ( top == null )
         throw new IllegalStateException("Can't pop from an empty stack.");
      int topItem = top.item;  // The item that is being popped.
      top = top.next;          // The previous second item is now on top.
      return topItem;
   }

   /**
    * Returns true if the stack is empty.  Returns false
    * if there are one or more items on the stack.
    */
   public boolean isEmpty() {
      return (top == null);
   }

} // end class StackOfInts
```

You should make sure that you understand how the `push` and `pop` operations operate on the linked list. Drawing some pictures might help. Note that the linked list is part of the `private` implementation of the *StackOfInts* class. A program that uses this class doesn't even need to know that a linked list is being used.

Now, it's pretty easy to implement a stack as an array instead of as a linked list. Since the number of items on the stack varies with time, a counter is needed to keep track of how many spaces in the array are actually in use. If this counter is called `top`, then the items on the stack are stored in positions `0`, `1`, ..., `top-1` in the array. The item in position `0` is on the bottom of the stack, and the item in position `top-1` is on the top of the stack. Pushing an item onto the stack is easy: Put the item in position `top` and add `1` to the value of `top`. If we don't want to put a limit on the number of items that the stack can hold, we can use the dynamic array techniques from Subsection 7.3.2. Note that the typical picture of the array would show the stack "upside down," with the bottom of the stack at the top of the array. This doesn't matter. The array is just an implementation of the abstract idea of a stack, and as long as the stack operations work the way they are supposed to, we are OK. Here is a second implementation of the *StackOfInts* class, using a dynamic array:

```
public class StackOfInts {  // (alternate version, using an array)

   private int[] items = new int[10];  // Holds the items on the stack.

   private int top = 0;  // The number of items currently on the stack.

   /**
    * Add N to the top of the stack.
    */
   public void push( int N ) {
       if (top == items.length) {
              // The array is full, so make a new, larger array and
              // copy the current stack items into it.
           int[] newArray = new int[ 2*items.length ];
           System.arraycopy(items, 0, newArray, 0, items.length);
           items = newArray;
       }
       items[top] = N;  // Put N in next available spot.
       top++;           // Number of items goes up by one.
   }

   /**
    * Remove the top item from the stack, and return it.
    * Throws an IllegalStateException if the stack is empty when
    * this method is called.
    */
   public int pop() {
       if ( top == 0 )
          throw new IllegalStateException("Can't pop from an empty stack.");
       int topItem = items[top - 1];  // Top item in the stack.
       top--;    // Number of items on the stack goes down by one.
       return topItem;
   }

   /**
    * Returns true if the stack is empty.  Returns false
    * if there are one or more items on the stack.
    */
   public boolean isEmpty() {
      return (top == 0);
   }

} // end class StackOfInts
```

Once again, the implementation of the stack (as an array) is private to the class. The two versions of the *StackOfInts* class can be used interchangeably, since their public interfaces are identical.

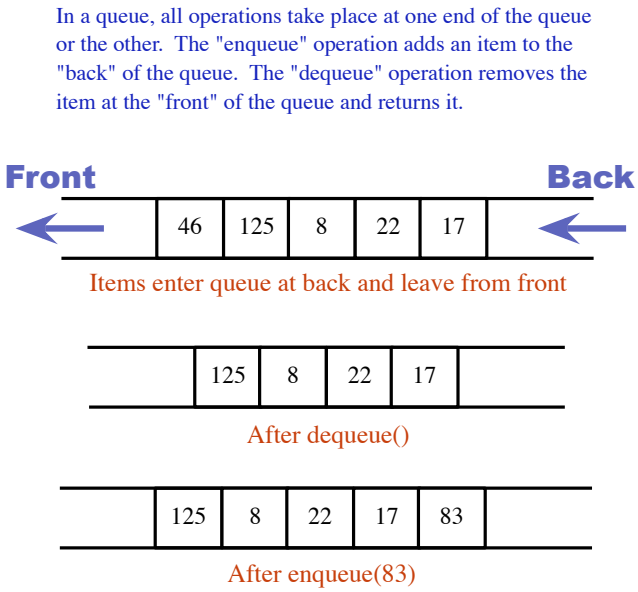
* * *

It's interesting to look at the run time analysis of stack operations. (See Section 8.5). We can measure the size of the problem by the number of items that are on the stack. For the linked list implementation of a stack, the worst case run time both for the `push` and for the `pop` operation is $\Theta(1)$. This just means that the run time is less than some constant, independent of the number of items on the stack. This is easy to see if you look at the code. The operations are implemented with a few simple assignment statements, and the number of items on the stack has no effect.

For the array implementation, on the other hand, a special case occurs in the `push` operation when the array is full. In that case, a new array is created and all the stack items are copied into the new array. This takes an amount of time that is proportional to the number of items on the stack. So, although the run time for `push` is usually $\Theta(1)$, the worst case run time is $\Theta(n)$, where n is the number of items on the stack. (However, the worst case occurs only rarely, and there is a natural sense in which the *average* case run time for the array implementation is still $\Theta(1)$.)

9.3.2 Queues

Queues are similar to stacks in that a queue consists of a sequence of items, and there are restrictions about how items can be added to and removed from the list. However, a queue has two ends, called the front and the back of the queue. Items are always added to the queue at the back and removed from the queue at the front. The operations of adding and removing items are called ***enqueue*** and ***dequeue***. An item that is added to the back of the queue will remain on the queue until all the items in front of it have been removed. This should sound familiar. A queue is like a "line" or "queue" of customers waiting for service. Customers are serviced in the order in which they arrive on the queue.



A queue can hold items of any type. For a queue of **ints**, the enqueue and dequeue operations can be implemented as instance methods in a "*QueueOfInts*" class. We also need an instance method for checking whether the queue is empty:

- `void enqueue(int N)` — Add N to the back of the queue.
- `int dequeue()` — Remove the item at the front and return it.
- `boolean isEmpty()` — Return true if the queue is empty.

A queue can be implemented as a linked list or as an array. An efficient array implementation is a little trickier than the array implementation of a stack, so I won't give it here. In the linked list implementation, the first item of the list is at the front of the queue. Dequeueing an item

from the front of the queue is just like popping an item off a stack. The back of the queue is at the end of the list. Enqueueing an item involves setting a pointer in the last node of the current list to point to a new node that contains the item. To do this, we'll need a command like "`tail.next = newNode;`", where `tail` is a pointer to the last node in the list. If `head` is a pointer to the first node of the list, it would always be possible to get a pointer to the last node of the list by saying:

```
Node tail;    // This will point to the last node in the list.
tail = head;  // Start at the first node.
while (tail.next != null) {
   tail = tail.next;  // Move to next node.
}
// At this point, tail.next is null, so tail points to
// the last node in the list.
```

However, it would be very inefficient to do this over and over every time an item is enqueued. For the sake of efficiency, we'll keep a pointer to the last node in an instance variable. This complicates the class somewhat; we have to be careful to update the value of this variable whenever a new node is added to the end of the list. Given all this, writing the *QueueOfInts* class is not all that difficult:

```
public class QueueOfInts {

   /**
    * An object of type Node holds one of the items
    * in the linked list that represents the queue.
    */
   private static class Node {
      int item;
      Node next;
   }

   private Node head = null;  // Points to first Node in the queue.
                              // The queue is empty when head is null.

   private Node tail = null;  // Points to last Node in the queue.

   /**
    * Add N to the back of the queue.
    */
   public void enqueue( int N ) {
      Node newTail = new Node();  // A Node to hold the new item.
      newTail.item = N;
      if (head == null) {
            // The queue was empty.  The new Node becomes
            // the only node in the list.  Since it is both
            // the first and last node, both head and tail
            // point to it.
         head = newTail;
         tail = newTail;
      }
      else {
            // The new node becomes the new tail of the list.
            // (The head of the list is unaffected.)
         tail.next = newTail;
```

```
            tail = newTail;
         }
      }

      /**
       * Remove and return the front item in the queue.
       * Throws an IllegalStateException if the queue is empty.
       */
      public int dequeue() {
         if ( head == null)
             throw new IllegalStateException("Can't dequeue from an empty queue.");
         int firstItem = head.item;
         head = head.next;  // The previous second item is now first.
         if (head == null) {
               // The queue has become empty.  The Node that was
               // deleted was the tail as well as the head of the
               // list, so now there is no tail.  (Actually, the
               // class would work fine without this step.)
            tail = null;
         }
         return firstItem;
      }

      /**
       * Return true if the queue is empty.
       */
      boolean isEmpty() {
         return (head == null);
      }

   } // end class QueueOfInts
```

Queues are typically used in a computer (as in real life) when only one item can be processed at a time, but several items can be waiting for processing. For example:

- In a Java program that has multiple threads, the threads that want processing time on the CPU are kept in a queue. When a new thread is started, it is added to the back of the queue. A thread is removed from the front of the queue, given some processing time, and then—if it has not terminated—is sent to the back of the queue to wait for another turn.
- Events such as keystrokes and mouse clicks are stored in a queue called the "event queue". A program removes events from the event queue and processes them. It's possible for several more events to occur while one event is being processed, but since the events are stored in a queue, they will always be processed in the order in which they occurred.
- A web server is a program that receives requests from web browsers for "pages." It is easy for new requests to arrive while the web server is still fulfilling a previous request. Requests that arrive while the web server is busy are placed into a queue to await processing. Using a queue ensures that requests will be processed in the order in which they were received.

Queues are said to implement a ***FIFO*** policy: First In, First Out. Or, as it is more commonly expressed, first come, first served. Stacks, on the other hand implement a ***LIFO*** policy: Last In, First Out. The item that comes out of the stack is the last one that was put in. Just like queues, stacks can be used to hold items that are waiting for processing (although in applications where queues are typically used, a stack would be considered "unfair").

* * *

To get a better handle on the difference between stacks and queues, consider the sample program *DepthBreadth.java*. I suggest that you run the program or try the applet version that can be found in the on-line version of this section. The program shows a grid of squares. Initially, all the squares are white. When you click on a white square, the program will gradually mark all the squares in the grid, starting from the one where you click. To understand how the program does this, think of yourself in the place of the program. When the user clicks a square, you are handed an index card. The location of the square—its row and column—is written on the card. You put the card in a pile, which then contains just that one card. Then, you repeat the following: If the pile is empty, you are done. Otherwise, remove an index card from the pile. The index card specifies a square. Look at each horizontal and vertical neighbor of that square. If the neighbor has not already been encountered, write its location on a new index card and put the card in the pile.

While a square is in the pile, waiting to be processed, it is colored red; that is, red squares have been *encountered* but not yet *processed*. When a square is taken from the pile and processed, its color changes to gray. Once a square has been colored gray, its color won't change again. Eventually, all the squares have been processed, and the procedure ends. In the index card analogy, the pile of cards has been emptied.

The program can use your choice of three methods: Stack, Queue, and Random. In each case, the same general procedure is used. The only difference is how the "pile of index cards" is managed. For a stack, cards are added and removed at the top of the pile. For a queue, cards are added to the bottom of the pile and removed from the top. In the random case, the card to be processed is picked at random from among all the cards in the pile. The order of processing is very different in these three cases.

You should experiment with the program to see how it all works. Try to understand how stacks and queues are being used. Try starting from one of the corner squares. While the process is going on, you can click on other white squares, and they will be added to the pile. When you do this with a stack, you should notice that the square you click is processed immediately, and all the red squares that were already waiting for processing have to wait. On the other hand, if you do this with a queue, the square that you click will wait its turn until all the squares that were already in the pile have been processed.

* * *

Queues seem very natural because they occur so often in real life, but there are times when stacks are appropriate and even essential. For example, consider what happens when a routine calls a subroutine. The first routine is suspended while the subroutine is executed, and it will continue only when the subroutine returns. Now, suppose that the subroutine calls a second subroutine, and the second subroutine calls a third, and so on. Each subroutine is suspended while the subsequent subroutines are executed. The computer has to keep track of all the subroutines that are suspended. It does this with a stack.

When a subroutine is called, an ***activation record*** is created for that subroutine. The activation record contains information relevant to the execution of the subroutine, such as its local variables and parameters. The activation record for the subroutine is placed on a stack. It will be removed from the stack and destroyed when the subroutine returns. If the subroutine calls another subroutine, the activation record of the second subroutine is pushed onto the stack, on top of the activation record of the first subroutine. The stack can continue to grow as more subroutines are called, and it shrinks as those subroutines return.

9.3.3 Postfix Expressions

As another example, stacks can be used to evaluate ***postfix expressions***. An ordinary mathematical expression such as `2+(15-12)*17` is called an ***infix expression***. In an infix expression, an operator comes in between its two operands, as in "`2 + 2`". In a postfix expression, an operator comes after its two operands, as in "`2 2 +`". The infix expression "`2+(15-12)*17`" would be written in postfix form as "`2 15 12 - 17 * +`". The "`-`" operator in this expression applies to the two operands that precede it, namely "`15`" and "`12`". The "`*`" operator applies to the two operands that precede it, namely "`15 12 -`" and "`17`". And the "`+`" operator applies to "`2`" and "`15 12 - 17 *`". These are the same computations that are done in the original infix expression.

Now, suppose that we want to process the expression "`2 15 12 - 17 * +`", from left to right and find its value. The first item we encounter is the `2`, but what can we do with it? At this point, we don't know what operator, if any, will be applied to the `2` or what the other operand might be. We have to remember the `2` for later processing. We do this by pushing it onto a stack. Moving on to the next item, we see a `15`, which is pushed onto the stack on top of the `2`. Then the `12` is added to the stack. Now, we come to the operator, "`-`". This operation applies to the two operands that preceded it in the expression. We have saved those two operands on the stack. So, to process the "`-`" operator, we pop two numbers from the stack, `12` and `15`, and compute `15 - 12` to get the answer `3`. This `3` must be remembered to be used in later processing, so we push it onto the stack, on top of the `2` that is still waiting there. The next item in the expression is a `17`, which is processed by pushing it onto the stack, on top of the `3`. To process the next item, "`*`", we pop two numbers from the stack. The numbers are `17` and the `3` that represents the value of "`15 12 -`". These numbers are multiplied, and the result, `51` is pushed onto the stack. The next item in the expression is a "`+`" operator, which is processed by popping `51` and `2` from the stack, adding them, and pushing the result, `53`, onto the stack. Finally, we've come to the end of the expression. The number on the stack is the value of the entire expression, so all we have to do is pop the answer from the stack, and we are done! The value of the expression is `53`.

Although it's easier for people to work with infix expressions, postfix expressions have some advantages. For one thing, postfix expressions don't require parentheses or precedence rules. The order in which operators are applied is determined entirely by the order in which they occur in the expression. This allows the algorithm for evaluating postfix expressions to be fairly straightforward:

```
Start with an empty stack
for each item in the expression:
    if the item is a number:
       Push the number onto the stack
    else if the item is an operator:
       Pop the operands from the stack  // Can generate an error
       Apply the operator to the operands
       Push the result onto the stack
    else
       There is an error in the expression
Pop a number from the stack  // Can generate an error
if the stack is not empty:
   There is an error in the expression
else:
   The last number that was popped is the value of the expression
```

Errors in an expression can be detected easily. For example, in the expression "`2 3 + *`", there are not enough operands for the "`*`" operation. This will be detected in the algorithm when an attempt is made to pop the second operand for "`*`" from the stack, since the stack will be empty. The opposite problem occurs in "`2 3 4 +`". There are not enough operators for all the numbers. This will be detected when the `2` is left still sitting in the stack at the end of the algorithm.

This algorithm is demonstrated in the sample program *PostfixEval.java*. This program lets you type in postfix expressions made up of non-negative real numbers and the operators "`+`", "`-`", "`*`", "`/`", and "`^`". The "`^`" represents exponentiation. That is, "`2 3 ^`" is evaluated as 2^3. The program prints out a message as it processes each item in the expression. The stack class that is used in the program is defined in the file *StackOfDouble.java*. The *StackOfDouble* class is identical to the first *StackOfInts* class, given above, except that it has been modified to store values of type **double** instead of values of type **int**.

The only interesting aspect of this program is the method that implements the postfix evaluation algorithm. It is a direct implementation of the pseudocode algorithm given above:

```
/**
 *  Read one line of input and process it as a postfix expression.
 *  If the input is not a legal postfix expression, then an error
 *  message is displayed.  Otherwise, the value of the expression
 *  is displayed.  It is assumed that the first character on
 *  the input line is a non-blank.
 */
private static void readAndEvaluate() {

   StackOfDouble stack;  // For evaluating the expression.

   stack = new StackOfDouble();  // Make a new, empty stack.

   TextIO.putln();

   while (TextIO.peek() != '\n') {

      if ( Character.isDigit(TextIO.peek()) ) {
             // The next item in input is a number.  Read it and
             // save it on the stack.
         double num = TextIO.getDouble();
         stack.push(num);
         TextIO.putln("   Pushed constant " + num);
      }
      else {
             // Since the next item is not a number, the only thing
             // it can legally be is an operator.  Get the operator
             // and perform the operation.
         char op;  // The operator, which must be +, -, *, /, or ^.
         double x,y;     // The operands, from the stack, for the operation.
         double answer;  // The result, to be pushed onto the stack.
         op = TextIO.getChar();
         if (op != '+' && op != '-' && op != '*' && op != '/' && op != '^') {
                // The character is not one of the acceptable operations.
            TextIO.putln("\nIllegal operator found in input: " + op);
            return;
         }
         if (stack.isEmpty()) {
```

```
                TextIO.putln("   Stack is empty while trying to evaluate " + op);
                TextIO.putln("\nNot enough numbers in expression!");
                return;
             }
             y = stack.pop();
             if (stack.isEmpty()) {
                TextIO.putln("   Stack is empty while trying to evaluate " + op);
                TextIO.putln("\nNot enough numbers in expression!");
                return;
             }
             x = stack.pop();
             switch (op) {
             case '+':
                answer = x + y;
                break;
             case '-':
                answer = x - y;
                break;
             case '*':
                answer = x * y;
                break;
             case '/':
                answer = x / y;
                break;
             default:
                answer = Math.pow(x,y);  // (op must be '^'.)
             }
             stack.push(answer);
             TextIO.putln("   Evaluated " + op + " and pushed " + answer);
          }

          TextIO.skipBlanks();

       }  // end while

       // If we get to this point, the input has been read successfully.
       // If the expression was legal, then the value of the expression is
       // on the stack, and it is the only thing on the stack.

       if (stack.isEmpty()) {  // Impossible if the input is really non-empty.
          TextIO.putln("No expression provided.");
          return;
       }

       double value = stack.pop();  // Value of the expression.
       TextIO.putln("   Popped " + value + " at end of expression.");

       if (stack.isEmpty() == false) {
          TextIO.putln("   Stack is not empty.");
          TextIO.putln("\nNot enough operators for all the numbers!");
          return;
       }

       TextIO.putln("\nValue = " + value);

    } // end readAndEvaluate()
```

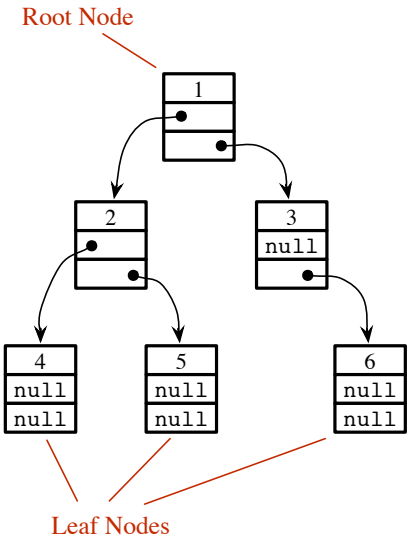
Postfix expressions are often used internally by computers. In fact, the Java virtual machine is a "stack machine" which uses the stack-based approach to expression evaluation that we have been discussing. The algorithm can easily be extended to handle variables, as well as constants. When a variable is encountered in the expression, the value of the variable is pushed onto the stack. It also works for operators with more or fewer than two operands. As many operands as are needed are popped from the stack and the result is pushed back onto the stack. For example, the ***unary minus*** operator, which is used in the expression "`-x`", has a single operand. We will continue to look at expressions and expression evaluation in the next two sections.

9.4 Binary Trees

We have seen in the two previous sections how objects can be linked into lists. When an object contains two pointers to objects of the same type, structures can be created that are much more complicated than linked lists. In this section, we'll look at one of the most basic and useful structures of this type: ***binary trees***. Each of the objects in a binary tree contains two pointers, typically called `left` and `right`. In addition to these pointers, of course, the nodes can contain other types of data. For example, a binary tree of integers could be made up of objects of the following type:

```
class TreeNode {
   int item;        // The data in this node.
   TreeNode left;   // Pointer to the left subtree.
   TreeNode right;  // Pointer to the right subtree.
}
```

The `left` and `right` pointers in a `TreeNode` can be `null` or can point to other objects of type `TreeNode`. A node that points to another node is said to be the ***parent*** of that node, and the node it points to is called a ***child***. In the picture below, for example, node 3 is the parent of node 6, and nodes 4 and 5 are children of node 2. Not every linked structure made up of tree nodes is a binary tree. A binary tree must have the following properties: There is exactly one node in the tree which has no parent. This node is called the ***root*** of the tree. Every other node in the tree has exactly one parent. Finally, there can be no loops in a binary tree. That is, it is not possible to follow a chain of pointers starting at some node and arriving back at the same node.



A node that has no children is called a ***leaf***. A leaf node can be recognized by the fact that both the left and right pointers in the node are `null`. In the standard picture of a binary tree, the root node is shown at the top and the leaf nodes at the bottom—which doesn't show much respect for the analogy to real trees. But at least you can see the branching, tree-like structure that gives a binary tree its name.

9.4.1 Tree Traversal

Consider any node in a binary tree. Look at that node together with all its descendants (that is, its children, the children of its children, and so on). This set of nodes forms a binary tree, which is called a ***subtree*** of the original tree. For example, in the picture, nodes 2, 4, and 5 form a subtree. This subtree is called the ***left subtree*** of the root. Similarly, nodes 3 and 6 make up the ***right subtree*** of the root. We can consider any non-empty binary tree to be made up of a root node, a left subtree, and a right subtree. Either or both of the subtrees can be empty. This is a recursive definition, matching the recursive definition of the *TreeNode* class. So it should not be a surprise that recursive subroutines are often used to process trees.

Consider the problem of counting the nodes in a binary tree. (As an exercise, you might try to come up with a non-recursive algorithm to do the counting, but you shouldn't expect to find one easily.) The heart of the problem is keeping track of which nodes remain to be counted. It's not so easy to do this, and in fact it's not even possible without an auxiliary data structure such as a stack or queue. With recursion, however, the algorithm is almost trivial. Either the tree is empty or it consists of a root and two subtrees. If the tree is empty, the number of nodes is zero. (This is the base case of the recursion.) Otherwise, use recursion to count the nodes in each subtree. Add the results from the subtrees together, and add one to count the root. This gives the total number of nodes in the tree. Written out in Java:

```
/**
 * Count the nodes in the binary tree to which root points, and
 * return the answer.  If root is null, the answer is zero.
 */
static int countNodes( TreeNode root ) {
   if ( root == null )
```

```
      return 0;  // The tree is empty.  It contains no nodes.
   else {
      int count = 1;   // Start by counting the root.
      count += countNodes(root.left);  // Add the number of nodes
                                       //     in the left subtree.
      count += countNodes(root.right); // Add the number of nodes
                                       //    in the right subtree.
      return count;  // Return the total.
   }
} // end countNodes()
```

Or, consider the problem of printing the items in a binary tree. If the tree is empty, there is nothing to do. If the tree is non-empty, then it consists of a root and two subtrees. Print the item in the root and use recursion to print the items in the subtrees. Here is a subroutine that prints all the items on one line of output:

```
/**
 * Print all the items in the tree to which root points.
 * The item in the root is printed first, followed by the
 * items in the left subtree and then the items in the
 * right subtree.
 */
static void preorderPrint( TreeNode root ) {
   if ( root != null ) {  // (Otherwise, there's nothing to print.)
      System.out.print( root.item + " " );  // Print the root item.
      preorderPrint( root.left );   // Print items in left subtree.
      preorderPrint( root.right );  // Print items in right subtree.
   }
} // end preorderPrint()
```

This routine is called "preorderPrint" because it uses a ***preorder traversal*** of the tree. In a preorder traversal, the root node of the tree is processed first, then the left subtree is traversed, then the right subtree. In a ***postorder traversal***, the left subtree is traversed, then the right subtree, and then the root node is processed. And in an ***inorder traversal***, the left subtree is traversed first, then the root node is processed, then the right subtree is traversed. Printing subroutines that use postorder and inorder traversal differ from `preorderPrint` only in the placement of the statement that outputs the root item:

```
/**
 * Print all the items in the tree to which root points.
 * The item in the left subtree is printed first, followed
 * by the items in the right subtree and then the item
 * in the root node.
 */
static void postorderPrint( TreeNode root ) {
   if ( root != null ) {  // (Otherwise, there's nothing to print.)
      postorderPrint( root.left );   // Print items in left subtree.
      postorderPrint( root.right );  // Print items in right subtree.
      System.out.print( root.item + " " );  // Print the root item.
   }
} // end postorderPrint()


/**
 * Print all the items in the tree to which root points.
```

```
 * The item in the left subtree is printed first, followed
 * by the item in the root node and then the items
 * in the right subtree.
 */
static void inorderPrint( TreeNode root ) {
   if ( root != null ) {  // (Otherwise, there's nothing to print.)
      inorderPrint( root.left );   // Print items in left subtree.
      System.out.print( root.item + " " );  // Print the root item.
      inorderPrint( root.right );  // Print items in right subtree.
   }
} // end inorderPrint()
```

Each of these subroutines can be applied to the binary tree shown in the illustration at the beginning of this section. The order in which the items are printed differs in each case:

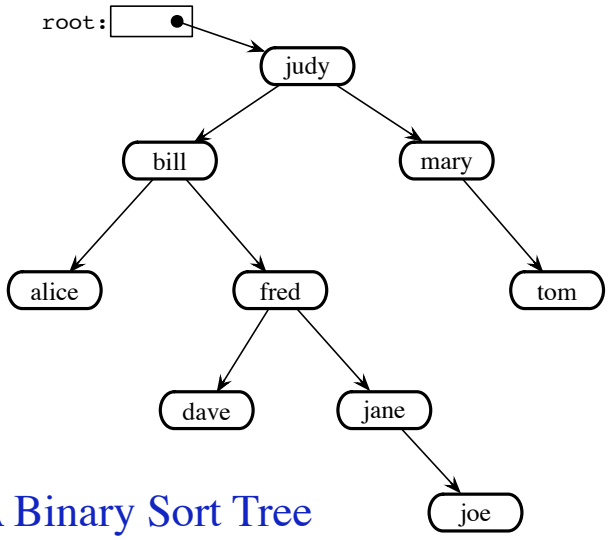
```
preorderPrint outputs:   1  2  4  5  3  6

postorderPrint outputs:  4  5  2  6  3  1

inorderPrint outputs:    4  2  5  1  3  6
```

In `preorderPrint`, for example, the item at the root of the tree, `1`, is output before anything else. But the preorder printing also applies to each of the subtrees of the root. The root item of the left subtree, `2`, is printed before the other items in that subtree, `4` and `5`. As for the right subtree of the root, `3` is output before `6`. A preorder traversal applies at all levels in the tree. The other two traversal orders can be analyzed similarly.

9.4.2 Binary Sort Trees

One of the examples in Section 9.2 was a linked list of strings, in which the strings were kept in increasing order. While a linked list works well for a small number of strings, it becomes inefficient for a large number of items. When inserting an item into the list, searching for that item's position requires looking at, on average, half the items in the list. Finding an item in the list requires a similar amount of time. If the strings are stored in a sorted array instead of in a linked list, then searching becomes more efficient because binary search can be used. However, inserting a new item into the array is still inefficient since it means moving, on average, half of the items in the array to make a space for the new item. A binary tree can be used to store an ordered list of strings, or other items, in a way that makes both searching and insertion efficient. A binary tree used in this way is called a ***binary sort tree.***

A binary sort tree is a binary tree with the following property: For every node in the tree, the item in that node is greater than every item in the left subtree of that node, and it is less than or equal to all the items in the right subtree of that node. Here for example is a binary sort tree containing items of type *String*. (In this picture, I haven't bothered to draw all the pointer variables. Non-null pointers are shown as arrows.)



A Binary Sort Tree

Binary sort trees have this useful property: An inorder traversal of the tree will process the items in increasing order. In fact, this is really just another way of expressing the definition. For example, if an inorder traversal is used to print the items in the tree shown above, then the items will be in alphabetical order. The definition of an inorder traversal guarantees that all the items in the left subtree of "judy" are printed before "judy", and all the items in the right subtree of "judy" are printed after "judy". But the binary sort tree property guarantees that the items in the left subtree of "judy" are precisely those that precede "judy" in alphabetical order, and all the items in the right subtree follow "judy" in alphabetical order. So, we know that "judy" is output in its proper alphabetical position. But the same argument applies to the subtrees. "Bill" will be output after "alice" and before "fred" and its descendents. "Fred" will be output after "dave" and before "jane" and "joe". And so on.

Suppose that we want to search for a given item in a binary search tree. Compare that item to the root item of the tree. If they are equal, we're done. If the item we are looking for is less than the root item, then we need to search the left subtree of the root—the right subtree can be eliminated because it only contains items that are greater than or equal to the root. Similarly, if the item we are looking for is greater than the item in the root, then we only need to look in the right subtree. In either case, the same procedure can then be applied to search the subtree. Inserting a new item is similar: Start by searching the tree for the position where the new item belongs. When that position is found, create a new node and attach it to the tree at that position.

Searching and inserting are efficient operations on a binary search tree, provided that the tree is close to being ***balanced***. A binary tree is balanced if for each node, the left subtree of that node contains approximately the same number of nodes as the right subtree. In a perfectly balanced tree, the two numbers differ by at most one. Not all binary trees are balanced, but if the tree is created by inserting items in a random order, there is a high probability that the tree is approximately balanced. (If the order of insertion is not random, however, it's quite possible for the tree to be very unbalanced.) During a search of any binary sort tree, every comparison eliminates one of two subtrees from further consideration. If the tree is balanced, that means cutting the number of items still under consideration in half. This is exactly the same as the binary search algorithm, and the result is a similarly efficient algorithm.

In terms of asymptotic analysis (Section 8.5), searching, inserting, and deleting in a binary

search tree have average case run time $\Theta(\log(n))$. The problem size, n, is the number of items in the tree, and the average is taken over all the different orders in which the items could have been inserted into the tree. As long as the actual insertion order is random, the actual run time can be expected to be close to the average. However, the worst case run time for binary search tree operations is $\Theta(n)$, which is much worse than $\Theta(\log(n))$. The worst case occurs for particular insertion orders. For example, if the items are inserted into the tree in order of increasing size, then every item that is inserted moves always to the right as it moves down the tree. The result is a "tree" that looks more like a linked list, since it consists of a linear string of nodes strung together by their `right` child pointers. Operations on such a tree have the same performance as operations on a linked list. Now, there are data structures that are similar to simple binary sort trees, except that insertion and deletion of nodes are implemented in a way that will always keep the tree balanced, or almost balanced. For these data structures, searching, inserting, and deleting have both average case and worst case run times that are $\Theta(\log(n))$. Here, however, we will look at only the simple versions of inserting and searching.

The sample program *SortTreeDemo.java* is a demonstration of binary sort trees. The program includes subroutines that implement inorder traversal, searching, and insertion. We'll look at the latter two subroutines below. The `main()` routine tests the subroutines by letting you type in strings to be inserted into the tree.

In this program, nodes in the binary tree are represented using the following static nested class, including a simple constructor that makes creating nodes easier:

```
/**
 * An object of type TreeNode represents one node in a binary tree of strings.
 */
private static class TreeNode {
   String item;      // The data in this node.
   TreeNode left;    // Pointer to left subtree.
   TreeNode right;   // Pointer to right subtree.
   TreeNode(String str) {
          // Constructor.  Make a node containing str.
      item = str;
   }
}  // end class TreeNode
```

A static member variable of type *TreeNode* points to the binary sort tree that is used by the program:

```
private static TreeNode root;  // Pointer to the root node in the tree.
                               // When the tree is empty, root is null.
```

A recursive subroutine named `treeContains` is used to search for a given item in the tree. This routine implements the search algorithm for binary trees that was outlined above:

```
/**
 * Return true if item is one of the items in the binary
 * sort tree to which root points.   Return false if not.
 */
static boolean treeContains( TreeNode root, String item ) {
   if ( root == null ) {
          // Tree is empty, so it certainly doesn't contain item.
      return false;
   }
   else if ( item.equals(root.item) ) {
```

```
            // Yes, the item has been found in the root node.
         return true;
      }
      else if ( item.compareTo(root.item) < 0 ) {
            // If the item occurs, it must be in the left subtree.
         return treeContains( root.left, item );
      }
      else {
            // If the item occurs, it must be in the right subtree.
         return treeContains( root.right, item );
      }
   }  // end treeContains()
```

When this routine is called in the `main()` routine, the first parameter is the static member variable `root`, which points to the root of the entire binary sort tree.

It's worth noting that recursion is not really essential in this case. A simple, non-recursive algorithm for searching a binary sort tree follows the rule: Start at the root and move down the tree until you find the item or reach a null pointer. Since the search follows a single path down the tree, it can be implemented as a `while` loop. Here is a non-recursive version of the search routine:

```
private static boolean treeContainsNR( TreeNode root, String item ) {
   TreeNode runner;  // For "running" down the tree.
   runner = root;    // Start at the root node.
   while (true) {
      if (runner == null) {
            // We've fallen off the tree without finding item.
         return false;
      }
      else if ( item.equals(node.item) ) {
            // We've found the item.
         return true;
      }
      else if ( item.compareTo(node.item) < 0 ) {
            // If the item occurs, it must be in the left subtree,
            // So, advance the runner down one level to the left.
         runner = runner.left;
      }
      else {
            // If the item occurs, it must be in the right subtree.
            // So, advance the runner down one level to the right.
         runner = runner.right;
      }
   }  // end while
} // end treeContainsNR();
```

The subroutine for inserting a new item into the tree turns out to be more similar to the non-recursive search routine than to the recursive. The insertion routine has to handle the case where the tree is empty. In that case, the value of `root` must be changed to point to a node that contains the new item:

```
root = new TreeNode( newItem );
```

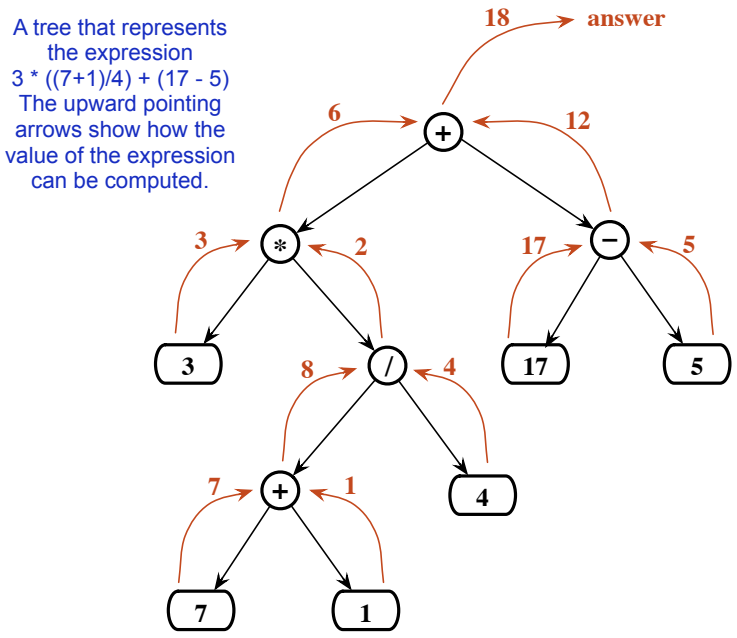
But this means, effectively, that the root can't be passed as a parameter to the subroutine, because it is impossible for a subroutine to change the value stored in an actual parameter. (I should note that this is something that **is** possible in other languages.) Recursion uses parameters in an essential way. There are ways to work around the problem, but the easiest thing is just to use a non-recursive insertion routine that accesses the static member variable `root` directly. One difference between inserting an item and searching for an item is that we have to be careful not to fall off the tree. That is, we have to stop searching just **before** `runner` becomes `null`. When we get to an empty spot in the tree, that's where we have to insert the new node:

```
/**
 * Add the item to the binary sort tree to which the global variable
 * "root" refers.  (Note that root can't be passed as  a parameter to
 * this routine because the value of root might change, and a change
 * in the value of a formal parameter does not change the actual parameter.)
 */
private static void treeInsert(String newItem) {
   if ( root == null ) {
          // The tree is empty.  Set root to point to a new node containing
          // the new item.  This becomes the only node in the tree.
      root = new TreeNode( newItem );
      return;
   }
   TreeNode runner;  // Runs down the tree to find a place for newItem.
   runner = root;   // Start at the root.
   while (true) {
      if ( newItem.compareTo(runner.item) < 0 ) {
             // Since the new item is less than the item in runner,
             // it belongs in the left subtree of runner.  If there
             // is an open space at runner.left, add a new node there.
             // Otherwise, advance runner down one level to the left.
         if ( runner.left == null ) {
            runner.left = new TreeNode( newItem );
            return;  // New item has been added to the tree.
         }
         else
            runner = runner.left;
      }
      else {
             // Since the new item is greater than or equal to the item in
             // runner, it belongs in the right subtree of runner.  If there
             // is an open space at runner.right, add a new node there.
             // Otherwise, advance runner down one level to the right.
         if ( runner.right == null ) {
            runner.right = new TreeNode( newItem );
            return;  // New item has been added to the tree.
         }
         else
            runner = runner.right;
      }
   } // end while
}  // end treeInsert()
```

9.4.3 Expression Trees

Another application of trees is to store mathematical expressions such as `15*(x+y)` or `sqrt(42)+7` in a convenient form. Let's stick for the moment to expressions made up of numbers and the operators `+`, `-`, `*`, and `/`. Consider the expression `3*((7+1)/4)+(17-5)`. This expression is made up of two subexpressions, `3*((7+1)/4)` and `(17-5)`, combined with the operator "+". When the expression is represented as a binary tree, the root node holds the operator `+`, while the subtrees of the root node represent the subexpressions `3*((7+1)/4)` and `(17-5)`. Every node in the tree holds either a number or an operator. A node that holds a number is a leaf node of the tree. A node that holds an operator has two subtrees representing the operands to which the operator applies. The tree is shown in the illustration below. I will refer to a tree of this type as an ***expression tree***.

Given an expression tree, it's easy to find the value of the expression that it represents. Each node in the tree has an associated value. If the node is a leaf node, then its value is simply the number that the node contains. If the node contains an operator, then the associated value is computed by first finding the values of its child nodes and then applying the operator to those values. The process is shown by the upward-directed arrows in the illustration. The value computed for the root node is the value of the expression as a whole. There are other uses for expression trees. For example, a postorder traversal of the tree will output the postfix form of the expression.



An expression tree contains two types of nodes: nodes that contain numbers and nodes that contain operators. Furthermore, we might want to add other types of nodes to make the trees more useful, such as nodes that contain variables. If we want to work with expression trees in Java, how can we deal with this variety of nodes? One way—which will be frowned upon by object-oriented purists—is to include an instance variable in each node object to record which type of node it is:

```
enum NodeType { NUMBER, OPERATOR }   // Possible kinds of node.
```

```
class ExpNode {  // A node in an expression tree.

    NodeType kind;  // Which type of node is this?
    double number;  // The value in a node of type NUMBER.
    char op;        // The operator in a node of type OPERATOR.
    ExpNode left;   // Pointers to subtrees,
    ExpNode right;  //     in a node of type OPERATOR.

    ExpNode( double val ) {
          // Constructor for making a node of type NUMBER.
       kind = NodeType.NUMBER;
       number = val;
    }

    ExpNode( char op, ExpNode left, ExpNode right ) {
          // Constructor for making a node of type OPERATOR.
       kind = NodeType.OPERATOR;
       this.op = op;
       this.left = left;
       this.right = right;
    }

 } // end class ExpNode
```

Given this definition, the following recursive subroutine will find the value of an expression tree:

```
static double getValue( ExpNode node ) {
       // Return the value of the expression represented by
       // the tree to which node refers.  Node must be non-null.
    if ( node.kind == NodeType.NUMBER ) {
          // The value of a NUMBER node is the number it holds.
       return node.number;
    }
    else {  // The kind must be OPERATOR.
            // Get the values of the operands and combine them
            //    using the operator.
       double leftVal = getValue( node.left );
       double rightVal = getValue( node.right );
       switch ( node.op ) {
          case '+':  return leftVal + rightVal;
          case '-':  return leftVal - rightVal;
          case '*':  return leftVal * rightVal;
          case '/':  return leftVal / rightVal;
          default:   return Double.NaN;  // Bad operator.
       }
    }
 } // end getValue()
```

Although this approach works, a more object-oriented approach is to note that since there are two types of nodes, there should be two classes to represent them, *ConstNode* and *BinOpNode*. To represent the general idea of a node in an expression tree, we need another class, *ExpNode*. Both *ConstNode* and *BinOpNode* will be subclasses of *ExpNode*. Since any actual node will be either a `ConstNode` or a *BinOpNode*, *ExpNode* should be an abstract class. (See Subsection 5.5.5.) Since one of the things we want to do with nodes is find their values, each class should have an instance method for finding the value:

```
abstract class ExpNode {
       // Represents a node of any type in an expression tree.

    abstract double value();  // Return the value of this node.

} // end class ExpNode


class ConstNode extends ExpNode {
       // Represents a node that holds a number.

    double number;  // The number in the node.

    ConstNode( double val ) {
          // Constructor.  Create a node to hold val.
       number = val;
    }

    double value() {
          // The value is just the number that the node holds.
       return number;
    }

 } // end class ConstNode


 class BinOpNode extends ExpNode {
       // Represents a node that holds an operator.

    char op;        // The operator.
    ExpNode left;   // The left operand.
    ExpNode right;  // The right operand.

    BinOpNode( char op, ExpNode left, ExpNode right ) {
          // Constructor.  Create a node to hold the given data.
       this.op = op;
       this.left = left;
       this.right = right;
    }

    double value() {
          // To get the value, compute the value of the left and
          // right operands, and combine them with the operator.
        double leftVal = left.value();
        double rightVal = right.value();
        switch ( op ) {
            case '+':  return leftVal + rightVal;
            case '-':  return leftVal - rightVal;
            case '*':  return leftVal * rightVal;
            case '/':  return leftVal / rightVal;
            default:   return Double.NaN;  // Bad operator.
         }
    }

 } // end class BinOpNode
```

Note that the left and right operands of a *BinOpNode* are of type *ExpNode*, not *BinOpNode*. This allows the operand to be either a *ConstNode* or another *BinOpNode*—or any other type of *ExpNode* that we might eventually create. Since every *ExpNode* has a `value()` method, we can

call `left.value()` to compute the value of the left operand. If `left` is in fact a *ConstNode*, this will call the `value()` method in the *ConstNode* class. If it is in fact a *BinOpNode*, then `left.value()` will call the `value()` method in the *BinOpNode* class. Each node knows how to compute its own value.

Although it might seem more complicated at first, the object-oriented approach has some advantages. For one thing, it doesn't waste memory. In the original *ExpNode* class, only some of the instance variables in each node were actually used, and we needed an extra instance variable to keep track of the type of node. More important, though, is the fact that new types of nodes can be added more cleanly, since it can be done by creating a new subclass of *ExpNode* rather than by modifying an existing class.

We'll return to the topic of expression trees in the next section, where we'll see how to create an expression tree to represent a given expression.

9.5 A Simple Recursive Descent Parser

I HAVE ALWAYS been fascinated by language—both natural languages like English and the artificial languages that are used by computers. There are many difficult questions about how languages can convey information, how they are structured, and how they can be processed. Natural and artificial languages are similar enough that the study of programming languages, which are pretty well understood, can give some insight into the much more complex and difficult natural languages. And programming languages raise more than enough interesting issues to make them worth studying in their own right. How can it be, after all, that computers can be made to "understand" even the relatively simple languages that are used to write programs? Computers can only directly use instructions expressed in very simple machine language. Higher level languages must be translated into machine language. But the translation is done by a compiler, which is just a program. How could such a translation program be written?

9.5.1 Backus-Naur Form

Natural and artificial languages are similar in that they have a structure known as grammar or syntax. Syntax can be expressed by a set of rules that describe what it means to be a legal sentence or program. For programming languages, syntax rules are often expressed in ***BNF*** (Backus-Naur Form), a system that was developed by computer scientists John Backus and Peter Naur in the late 1950s. Interestingly, an equivalent system was developed independently at about the same time by linguist Noam Chomsky to describe the grammar of natural language. BNF cannot express all possible syntax rules. For example, it can't express the fact that a variable must be defined before it is used. Furthermore, it says nothing about the meaning or semantics of the language. The problem of specifying the semantics of a language—even of an artificial programming language—is one that is still far from being completely solved. However, BNF does express the basic structure of the language, and it plays a central role in the design of translation programs.

In English, terms such as "noun", "transitive verb," and "prepositional phrase" are ***syntactic categories*** that describe building blocks of sentences. Similarly, "statement", "number," and "while loop" are syntactic categories that describe building blocks of Java programs. In BNF, a syntactic category is written as a word enclosed between "<" and ">". For example: `<noun>`, `<verb-phrase>`, or `<while-loop>`. A ***rule*** in BNF specifies the structure of an item in a given syntactic category, in terms of other syntactic categories and/or basic symbols of the

language. For example, one BNF rule for the English language might be

```
<sentence>  ::=  <noun-phrase> <verb-phrase>
```

The symbol "`::=`" is read "can be", so this rule says that a `<sentence>` can be a `<noun-phrase>` followed by a `<verb-phrase>`. (The term is "can be" rather than "is" because there might be other rules that specify other possible forms for a sentence.) This rule can be thought of as a recipe for a sentence: If you want to make a sentence, make a noun-phrase and follow it by a verb-phrase. Noun-phrase and verb-phrase must, in turn, be defined by other BNF rules.

In BNF, a choice between alternatives is represented by the symbol "`|`", which is read "or". For example, the rule

```
<verb-phrase>  ::=  <intransitive-verb>  |
                    ( <transitive-verb> <noun-phrase> )
```

says that a `<verb-phrase>` can be an `<intransitive-verb>`, or a `<transitive-verb>` followed by a `<noun-phrase>`. Note also that parentheses can be used for grouping. To express the fact that an item is optional, it can be enclosed between "`[`" and "`]`". An optional item that can be repeated any number of times is enclosed between "`[`" and "`]...`". And a symbol that is an actual part of the language that is being described is enclosed in quotes. For example,

```
<noun-phrase>  ::=  <common-noun> [ "that" <verb-phrase> ]  |
                    <common-noun> [ <prepositional-phrase> ]...
```

says that a `<noun-phrase>` can be a `<common-noun>`, optionally followed by the literal word "`that`" and a `<verb-phrase>`, or it can be a `<common-noun>` followed by zero or more `<prepositional-phrase>'s`. Obviously, we can describe very complex structures in this way. The real power comes from the fact that BNF rules can be **recursive**. In fact, the two preceding rules, taken together, are recursive. A `<noun-phrase>` is defined partly in terms of `<verb-phrase>`, while `<verb-phrase>` is defined partly in terms of `<noun-phrase>`. For example, a `<noun-phrase>` might be "the rat that ate the cheese", since "ate the cheese" is a `<verb-phrase>`. But then we can, recursively, make the more complex `<noun-phrase>` "the cat that caught the rat that ate the cheese" out of the `<common-noun>` "the cat", the word "that" and the `<verb-phrase>` "caught the rat that ate the cheese". Building from there, we can make the `<noun-phrase>` "the dog that chased the cat that caught the rat that ate the cheese". The recursive structure of language is one of the most fundamental properties of language, and the ability of BNF to express this recursive structure is what makes it so useful.

BNF can be used to describe the syntax of a programming language such as Java in a formal and precise way. For example, a `<while-loop>` can be defined as

```
<while-loop>  ::=  "while" "(" <condition> ")" <statement>
```

This says that a `<while-loop>` consists of the word "while", followed by a left parenthesis, followed by a `<condition>`, followed by a right parenthesis, followed by a `<statement>`. Of course, it still remains to define what is meant by a condition and by a statement. Since a statement can be, among other things, a `while` loop, we can already see the recursive structure of the Java language. The exact specification of an `if` statement, which is hard to express clearly in words, can be given as

```
<if-statement>  ::=
             "if" "(" <condition> ")" <statement>
             [ "else" "if" "(" <condition> ")" <statement> ]...
             [ "else" <statement> ]
```

This rule makes it clear that the "`else`" part is optional and that there can be, optionally, one or more "`else if`" parts.

9.5.2 Recursive Descent Parsing

In the rest of this section, I will show how a BNF grammar for a language can be used as a guide for constructing a parser. A parser is a program that determines the grammatical structure of a phrase in the language. This is the first step in determining the meaning of the phrase—which for a programming language means translating it into machine language. Although we will look at only a simple example, I hope it will be enough to convince you that compilers can in fact be written and understood by mortals and to give you some idea of how that can be done.

The parsing method that we will use is called ***recursive descent parsing***. It is not the only possible parsing method, or the most efficient, but it is the one most suited for writing compilers by hand (rather than with the help of so called "parser generator" programs). In a recursive descent parser, every rule of the BNF grammar is the model for a subroutine. Not every BNF grammar is suitable for recursive descent parsing. The grammar must satisfy a certain property. Essentially, while parsing a phrase, it must be possible to tell what syntactic category is coming up next just by looking at the next item in the input. Many grammars are designed with this property in mind.

I should also mention that many variations of BNF are in use. The one that I've described here is one that is well-suited for recursive descent parsing.

* * *

When we try to parse a phrase that contains a syntax error, we need some way to respond to the error. A convenient way of doing this is to throw an exception. I'll use an exception class called *ParseError*, defined as follows:

```
/**
 * An object of type ParseError represents a syntax error found in
 * the user's input.
 */
private static class ParseError extends Exception {
   ParseError(String message) {
      super(message);
   }
} // end nested class ParseError
```

Another general point is that our BNF rules don't say anything about spaces between items, but in reality we want to be able to insert spaces between items at will. To allow for this, I'll always call the routine `TextIO.skipBlanks()` before trying to look ahead to see what's coming up next in input. `TextIO.skipBlanks()` skips past any whitespace, such as spaces and tabs, in the input, and stops when the next character in the input is either a non-blank character or the end-of-line character.

Let's start with a very simple example. A "fully parenthesized expression" can be specified in BNF by the rules

```
<expression>  ::=  <number>  |
                   "(" <expression> <operator> <expression> ")"

<operator>  ::=  "+" | "-" | "*" | "/"
```

where <number> refers to any non-negative real number. An example of a fully parenthesized expression is "(((34-17)*8)+(2*7))". Since every operator corresponds to a pair of parentheses, there is no ambiguity about the order in which the operators are to be applied. Suppose we want a program that will read and evaluate such expressions. We'll read the expressions from standard input, using *TextIO*. To apply recursive descent parsing, we need a subroutine for each rule in the grammar. Corresponding to the rule for <operator>, we get a subroutine that reads an operator. The operator can be a choice of any of four things. Any other input will be an error.

```
/**
 * If the next character in input is one of the legal operators,
 * read it and return it.  Otherwise, throw a ParseError.
 */
static char getOperator() throws ParseError {
   TextIO.skipBlanks();
   char op = TextIO.peek(); // look ahead at the next char, without reading it
   if ( op == '+' || op == '-' || op == '*' || op == '/' ) {
      TextIO.getAnyChar();  // read the operator, to remove it from the input
      return op;
   }
   else if (op == '\n')
      throw new ParseError("Missing operator at end of line.");
   else
      throw new ParseError("Missing operator.  Found \"" +
            op + "\" instead of +, -, *, or /.");
} // end getOperator()
```

I've tried to give a reasonable error message, depending on whether the next character is an end-of-line or something else. I use `TextIO.peek()` to look ahead at the next character before I read it, and I call `TextIO.skipBlanks()` before testing `TextIO.peek()` in order to ignore any blanks that separate items. I will follow this same pattern in every case.

When we come to the subroutine for <expression>, things are a little more interesting. The rule says that an expression can be either a number or an expression enclosed in parentheses. We can tell which it is by looking ahead at the next character. If the character is a digit, we have to read a number. If the character is a "(", we have to read the "(", followed by an expression, followed by an operator, followed by another expression, followed by a ")". If the next character is anything else, there is an error. Note that we need recursion to read the nested expressions. The routine doesn't just read the expression. It also computes and returns its value. This requires semantical information that is not specified in the BNF rule.

```
/**
 * Read an expression from the current line of input and return its value.
 * @throws ParseError if the input contains a syntax error
 */
private static double expressionValue() throws ParseError {
   TextIO.skipBlanks();
   if ( Character.isDigit(TextIO.peek()) ) {
          // The next item in input is a number, so the expression
          // must consist of just that number.  Read and return
          // the number.
      return TextIO.getDouble();
   }
   else if ( TextIO.peek() == '(' ) {
```

```
            // The expression must be of the form
            //         "(" <expression> <operator> <expression> ")"
            // Read all these items, perform the operation, and
            // return the result.
         TextIO.getAnyChar();  // Read the "("
         double leftVal = expressionValue();  // Read and evaluate first operand.
         char op = getOperator();             // Read the operator.
         double rightVal = expressionValue(); // Read and evaluate second operand.
         TextIO.skipBlanks();
         if ( TextIO.peek() != ')' ) {
                // According to the rule, there must be a ")" here.
                // Since it's missing, throw a ParseError.
            throw new ParseError("Missing right parenthesis.");
         }
         TextIO.getAnyChar();  // Read the ")"
         switch (op) {   //  Apply the operator and return the result.
         case '+':  return leftVal + rightVal;
         case '-':  return leftVal - rightVal;
         case '*':  return leftVal * rightVal;
         case '/':  return leftVal / rightVal;
         default:   return 0;  // Can't occur since op is one of the above.
                               // (But Java syntax requires a return value.)
         }
      }
      else {  // No other character can legally start an expression.
         throw new ParseError("Encountered unexpected character, \"" +
               TextIO.peek() + "\" in input.");
      }
   } // end expressionValue()
```

I hope that you can see how this routine corresponds to the BNF rule. Where the rule uses "`|`" to give a choice between alternatives, there is an `if` statement in the routine to determine which choice to take. Where the rule contains a sequence of items, "(" `<expression>` `<operator>` `<expression>` ")", there is a sequence of statements in the subroutine to read each item in turn.

When `expressionValue()` is called to evaluate the expression `(((34-17)*8)+(2*7))`, it sees the "(" at the beginning of the input, so the `else` part of the `if` statement is executed. The "(" is read. Then the first recursive call to `expressionValue()` reads and evaluates the subexpression `((34-17)*8)`, the call to `getOperator()` reads the "+" operator, and the second recursive call to `expressionValue()` reads and evaluates the second subexpression `(2*7)`. Finally, the ")" at the end of the expression is read. Of course, reading the first subexpression, `((34-17)*8)`, involves further recursive calls to the `expressionValue()` routine, but it's better not to think too deeply about that! Rely on the recursion to handle the details.

You'll find a complete program that uses these routines in the file *SimpleParser1.java*.

* * *

Fully parenthesized expressions aren't very natural for people to use. But with ordinary expressions, we have to worry about the question of operator precedence, which tells us, for example, that the "`*`" in the expression "`5+3*7`" is applied before the "`+`". The complex expression "`3*6+8*(7+1)/4-24`" should be seen as made up of three "terms", `3*6`, `8*(7+1)/4`, and `24`, combined with "`+`" and "`-`" operators. A term, on the other hand, can be made up of several factors combined with "`*`" and "`/`" operators. For example, `8*(7+1)/4` contains the

factors `8`, `(7+1)` and `4`. This example also shows that a factor can be either a number or an expression in parentheses. To complicate things a bit more, we allow for leading minus signs in expressions, as in "`-(3+4)`" or "`-7`". (Since a `<number>` is a positive number, this is the only way we can get negative numbers. It's done this way to avoid "`3 * -7`", for example.) This structure can be expressed by the BNF rules

```
<expression>  ::=  [ "-" ] <term> [ ( "+" | "-" ) <term> ]...
<term>  ::=  <factor> [ ( "*" | "/" ) <factor> ]...
<factor>  ::=  <number>  |  "(" <expression> ")"
```

The first rule uses the "`[ ]...`" notation, which says that the items that it encloses can occur zero, one, two, or more times. This means that an `<expression>` can begin, optionally, with a "-". Then there must be a `<term>` which can optionally be followed by one of the operators "+" or "−" and another `<term>`, optionally followed by another operator and `<term>`, and so on. In a subroutine that reads and evaluates expressions, this repetition is handled by a `while` loop. An `if` statement is used at the beginning of the loop to test whether a leading minus sign is present:

```
/**
 * Read an expression from the current line of input and return its value.
 * @throws ParseError if the input contains a syntax error
 */
private static double expressionValue() throws ParseError {
   TextIO.skipBlanks();
   boolean negative;  // True if there is a leading minus sign.
   negative = false;
   if (TextIO.peek() == '-') {
      TextIO.getAnyChar();  // Read the minus sign.
      negative = true;
   }
   double val;  // Value of the expression.
   val = termValue();
   if (negative)
      val = -val;
   TextIO.skipBlanks();
   while ( TextIO.peek() == '+' || TextIO.peek() == '-' ) {
          // Read the next term and add it to or subtract it from
          // the value of previous terms in the expression.
      char op = TextIO.getAnyChar();  // Read the operator.
      double nextVal = termValue();
      if (op == '+')
         val += nextVal;
      else
         val -= nextVal;
      TextIO.skipBlanks();
   }
   return val;
} // end expressionValue()
```

The subroutine for `<term>` is very similar to this, and the subroutine for `<factor>` is similar to the example given above for fully parenthesized expressions. A complete program that reads and evaluates expressions based on the above BNF rules can be found in the file *SimpleParser2.java*.

9.5.3 Building an Expression Tree

Now, so far, we've only evaluated expressions. What does that have to do with translating programs into machine language? Well, instead of actually evaluating the expression, it would be almost as easy to generate the machine language instructions that are needed to evaluate the expression. If we are working with a "stack machine," these instructions would be stack operations such as "push a number" or "apply a + operation". The program *SimpleParser3.java* can both evaluate the expression and print a list of stack machine operations for evaluating the expression.

It's quite a jump from this program to a recursive descent parser that can read a program written in Java and generate the equivalent machine language code—but the conceptual leap is not huge.

The `SimpleParser3` program doesn't actually generate the stack operations directly as it parses an expression. Instead, it builds an expression tree, as discussed in Subsection 9.4.3, to represent the expression. The expression tree is then used to find the value and to generate the stack operations. The tree is made up of nodes belonging to classes *ConstNode* and *BinOpNode* that are similar to those given in Subsection 9.4.3. Another class, *UnaryMinusNode*, has been introduced to represent the unary minus operation. I've added a method, `printStackCommands()`, to each class. This method is responsible for printing out the stack operations that are necessary to evaluate an expression. Here for example is the new `BinOpNode` class from *SimpleParser3.java*:

```
private static class BinOpNode extends ExpNode {
   char op;        // The operator.
   ExpNode left;   // The expression for its left operand.
   ExpNode right;  // The expression for its right operand.
   BinOpNode(char op, ExpNode left, ExpNode right) {
          // Construct a BinOpNode containing the specified data.
      assert op == '+' || op == '-' || op == '*' || op == '/';
      assert left != null && right != null;
      this.op = op;
      this.left = left;
      this.right = right;
   }
   double value() {
          // The value is obtained by evaluating the left and right
          // operands and combining the values with the operator.
      double x = left.value();
      double y = right.value();
      switch (op) {
      case '+':
         return x + y;
      case '-':
         return x - y;
      case '*':
         return x * y;
      case '/':
         return x / y;
      default:
         return Double.NaN;  // Bad operator!
      }
   }
```

```
   void  printStackCommands() {
          // To evaluate the expression on a stack machine, first do
          // whatever is necessary to evaluate the left operand, leaving
          // the answer on the stack.  Then do the same thing for the
          // second operand.  Then apply the operator (which means popping
          // the operands, applying the operator, and pushing the result).
      left.printStackCommands();
      right.printStackCommands();
      TextIO.putln("  Operator " + op);
   }
}
```

It's also interesting to look at the new parsing subroutines. Instead of computing a value, each subroutine builds an expression tree. For example, the subroutine corresponding to the rule for `<expression>` becomes

```
static ExpNode expressionTree() throws ParseError {
       // Read an expression from the current line of input and
       // return an expression tree representing the expression.
   TextIO.skipBlanks();
   boolean negative;  // True if there is a leading minus sign.
   negative = false;
   if (TextIO.peek() == '-') {
      TextIO.getAnyChar();
      negative = true;
   }
   ExpNode exp;   // The expression tree for the expression.
   exp = termTree();  // Start with a tree for first term.
   if (negative) {
          // Build the tree that corresponds to applying a
          // unary minus operator to the term we've
          // just read.
      exp = new UnaryMinusNode(exp);
   }
   TextIO.skipBlanks();
   while ( TextIO.peek() == '+' || TextIO.peek() == '-' ) {
            // Read the next term and combine it with the
            // previous terms into a bigger expression tree.
       char op = TextIO.getAnyChar();
       ExpNode nextTerm = termTree();
            // Create a tree that applies the binary operator
            // to the previous tree and the term we just read.
       exp = new BinOpNode(op, exp, nextTerm);
       TextIO.skipBlanks();
   }
   return exp;
} // end expressionTree()
```

In some real compilers, the parser creates a tree to represent the program that is being parsed. This tree is called a ***parse tree***. Parse trees are somewhat different in form from expression trees, but the purpose is the same. Once you have the tree, there are a number of things you can do with it. For one thing, it can be used to generate machine language code. But

there are also techniques for examining the tree and detecting certain types of programming errors, such as an attempt to reference a local variable before it has been assigned a value. (The Java compiler, of course, will reject the program if it contains such an error.) It's also possible to manipulate the tree to ***optimize*** the program. In optimization, the tree is transformed to make the program more efficient before the code is generated.

And so we are back where we started in Chapter 1, looking at programming languages, compilers, and machine language. But looking at them, I hope, with a lot more understanding and a much wider perspective.

Exercises for Chapter 9

1. In many textbooks, the first examples of recursion are the mathematical functions *factorial* and *fibonacci*. These functions are defined for non-negative integers using the following recursive formulas:

```
factorial(0)  =  1
factorial(N)  =  N*factorial(N-1)   for N > 0

fibonacci(0)  =  1
fibonacci(1)  =  1
fibonacci(N)  =  fibonacci(N-1) + fibonacci(N-2)   for N > 1
```

Write recursive functions to compute `factorial(N)` and `fibonacci(N)` for a given non-negative integer `N`, and write a `main()` routine to test your functions.

(In fact, *factorial* and *fibonacci* are really not very good examples of recursion, since the most natural way to compute them is to use simple `for` loops. Furthermore, *fibonacci* is a particularly bad example, since the natural recursive approach to computing this function is extremely inefficient.)

2. Exercise 7.6 asked you to read a file, make an alphabetical list of all the words that occur in the file, and write the list to another file. In that exercise, you were asked to use an ArrayList<String> to store the words. Write a new version of the same program that stores the words in a binary sort tree instead of in an arraylist. You can use the binary sort tree routines from *SortTreeDemo.java*, which was discussed in Subsection 9.4.2.

3. Suppose that linked lists of integers are made from objects belonging to the class

```
class ListNode {
   int item;       // An item in the list.
   ListNode next;  // Pointer to the next node in the list.
}
```

Write a subroutine that will make a copy of a list, with the order of the items of the list reversed. The subroutine should have a parameter of type *ListNode*, and it should return a value of type *ListNode*. The original list should not be modified.

You should also write a `main()` routine to test your subroutine.

4. Subsection 9.4.1 explains how to use recursion to print out the items in a binary tree in various orders. That section also notes that a non-recursive subroutine can be used to print the items, provided that a stack or queue is used as an auxiliary data structure. Assuming that a queue is used, here is an algorithm for such a subroutine:

```
Add the root node to an empty queue
while the queue is not empty:
   Get a node from the queue
   Print the item in the node
   if node.left is not null:
      add it to the queue
   if node.right is not null:
      add it to the queue
```

Write a subroutine that implements this algorithm, and write a program to test the subroutine. Note that you will need a queue of *TreeNodes*, so you will need to write a class to represent such queues.

(Note that the order in which items are printed by this algorithm is different from all three of the orders considered in Subsection 9.4.1.)

5. In Subsection 9.4.2, I say that "if the [binary sort] tree is created by inserting items in a random order, there is a high probability that the tree is approximately balanced." For this exercise, you will do an experiment to test whether that is true.

The ***depth*** of a node in a binary tree is the length of the path from the root of the tree to that node. That is, the root has depth 0, its children have depth 1, its grandchildren have depth 2, and so on. In a balanced tree, all the leaves in the tree are about the same depth. For example, in a perfectly balanced tree with 1023 nodes, all the leaves are at depth 9. In an approximately balanced tree with 1023 nodes, the average depth of all the leaves should be not too much bigger than 9.

On the other hand, even if the tree is approximately balanced, there might be a few leaves that have much larger depth than the average, so we might also want to look at the maximum depth among all the leaves in a tree.

For this exercise, you should create a random binary sort tree with 1023 nodes. The items in the tree can be real numbers, and you can create the tree by generating 1023 random real numbers and inserting them into the tree, using the usual `treeInsert()` method for binary sort trees. Once you have the tree, you should compute and output the average depth of all the leaves in the tree and the maximum depth of all the leaves. To do this, you will need three recursive subroutines: one to count the leaves, one to find the sum of the depths of all the leaves, and one to find the maximum depth. The latter two subroutines should have an **int**-valued parameter, `depth`, that tells how deep in the tree you've gone. When you call this routine from the main program, the `depth` parameter is 0; when you call the routine recursively, the parameter increases by 1.

6. The parsing programs in Section 9.5 work with expressions made up of numbers and operators. We can make things a little more interesting by allowing the variable "x" to occur. This would allow expression such as "`3*(x-1)*(x+1)`", for example. Make a new version of the sample program *SimpleParser3.java* that can work with such expressions. In your program, the `main()` routine can't simply print the value of the expression, since the value of the expression now depends on the value of `x`. Instead, it should print the value of the expression for `x=0`, `x=1`, `x=2`, and `x=3`.

The original program will have to be modified in several other ways. Currently, the program uses classes *ConstNode*, *BinOpNode*, and *UnaryMinusNode* to represent nodes in an expression tree. Since expressions can now include `x`, you will need a new class, *VariableNode*, to represent an occurrence of `x` in the expression.

In the original program, each of the node classes has an instance method, "`double value()`", which returns the value of the node. But in your program, the value can depend on `x`, so you should replace this method with one of the form "`double value(double xValue)`", where the parameter `xValue` is the value of `x`.

Finally, the parsing subroutines in your program will have to take into account the fact that expressions can contain `x`. There is just one small change in the BNF rules for the expressions: A `<factor>` is allowed to be the variable `x`:

```
<factor>  ::=  <number>  |  <x-variable>  |  "(" <expression> ")"
```

where `<x-variable>` can be either a lower case or an upper case "X". This change in the BNF requires a change in the `factorTree()` subroutine.

7. This exercise builds on the previous exercise, Exercise 9.6. To understand it, you should have some background in Calculus. The derivative of an expression that involves the variable `x` can be defined by a few recursive rules:

- The derivative of a constant is 0.
- The derivative of `x` is 1.
- If `A` is an expression, let `dA` be the derivative of `A`. Then the derivative of `-A` is `-dA`.
- If `A` and `B` are expressions, let `dA` be the derivative of `A` and let `dB` be the derivative of `B`. Then the derivative of `A+B` is `dA+dB`.
- The derivative of `A-B` is `dA-dB`.
- The derivative of `A*B` is `A*dB + B*dA`.
- The derivative of `A/B` is `(B*dA - A*dB) / (B*B)`.

For this exercise, you should modify your program from the previous exercise so that it can compute the derivative of an expression. You can do this by adding a derivative-computing method to each of the node classes. First, add another abstract method to the *ExpNode* class:

```
abstract ExpNode derivative();
```

Then implement this method in each of the four subclasses of *ExpNode*. All the information that you need is in the rules given above. In your main program, instead of printing the stack operations for the original expression, you should print out the stack operations that define the derivative. Note that the formula that you get for the derivative can be much more complicated than it needs to be. For example, the derivative of `3*x+1` will be computed as `(3*1+0*x)+0`. This is correct, even though it's kind of ugly, and it would be nice for it to be simplified. However, simplifying expressions is not easy.

As an alternative to printing out stack operations, you might want to print the derivative as a fully parenthesized expression. You can do this by adding a `printInfix()` routine to each node class. It would be nice to leave out unnecessary parentheses, but again, the problem of deciding which parentheses can be left out without altering the meaning of the expression is a fairly difficult one, which I don't advise you to attempt.

(There is one curious thing that happens here: If you apply the rules, as given, to an expression tree, the result is no longer a tree, since the same subexpression can occur at multiple points in the derivative. For example, if you build a node to represent `B*B` by saying "`new BinOpNode('*',B,B)`", then the left and right children of the new node are actually the same node! This is not allowed in a tree. However, the difference is harmless in this case since, like a tree, the structure that you get has no loops in it. Loops, on the other hand, would be a disaster in most of the recursive tree-processing subroutines that we have written, since it would lead to infinite recursion. The type of structure that is built by the derivative functions is technically referred to as a ***directed acyclic graph***.)

Quiz on Chapter 9

1. Explain what is meant by a *recursive* subroutine.

2. Consider the following subroutine:

```
static void printStuff(int level) {
    if (level == 0) {
       System.out.print("*");
    }
    else {
       System.out.print("[");
       printStuff(level - 1);
       System.out.print(",");
       printStuff(level - 1);
       System.out.println("]");
    }
}
```

Show the output that would be produced by the subroutine calls `printStuff(0)`, `printStuff(1)`, `printStuff(2)`, and `printStuff(3)`.

3. Suppose that a linked list is formed from objects that belong to the class

```
class ListNode {
   int item;       // An item in the list.
   ListNode next;  // Pointer to next item in the list.
}
```

Write a subroutine that will count the number of zeros that occur in a given linked list of **ints**. The subroutine should have a parameter of type `ListNode` and should return a value of type **int**.

4. What are the three operations on a *stack?*

5. What is the basic difference between a stack and a queue?

6. What is an *activation record*? What role does a stack of activation records play in a computer?

7. Suppose that a binary tree of integers is formed from objects belonging to the class

```
class TreeNode {
   int item;       // One item in the tree.
   TreeNode left;  // Pointer to the left subtree.
   TreeNode right; // Pointer to the right subtree.
}
```

Write a recursive subroutine that will find the sum of all the nodes in the tree. Your subroutine should have a parameter of type `TreeNode`, and it should return a value of type **int**.

8. What is a *postorder traversal* of a binary tree?

9. Suppose that a `<multilist>` is defined by the BNF rule

```
<multilist>  ::=  <word>  |  "(" [ <multilist> ]... ")"
```

where a `<word>` can be any sequence of letters. Give five different `<multilist>'s` that can be generated by this rule. (This rule, by the way, is almost the entire syntax of the programming language `LISP`! `LISP` is known for its simple syntax and its elegant and powerful semantics.)

10. Explain what is meant by *parsing* a computer program.

Chapter 10

Generic Programming and Collection Classes

How to avoid reinventing the wheel? Many data structures and algorithms, such as those from Chapter 9, have been studied, programmed, and re-programmed by generations of computer science students. This is a valuable learning experience. Unfortunately, they have also been programmed and re-programmed by generations of working computer professionals, taking up time that could be devoted to new, more creative work. A programmer who needs a list or a binary tree shouldn't have to re-code these data structures from scratch. They are well-understood and have been programmed thousands of times before. The problem is how to make pre-written, robust data structures available to programmers. In this chapter, we'll look at Java's attempt to address this problem.

10.1 Generic Programming

Generic programming refers to writing code that will work for many types of data. We encountered the term in Section 7.3, where we looked at dynamic arrays of integers. The source code presented there for working with dynamic arrays of integers works only for data of type **int**. But the source code for dynamic arrays of **double**, *String*, *JButton*, or any other type would be almost identical, except for the substitution of one type name for another. It seems silly to write essentially the same code over and over. As we saw in Subsection 7.3.3, Java goes some distance towards solving this problem by providing the *ArrayList* class. An *ArrayList* is essentially a dynamic array of values of type *Object*. Since every class is a subclass of *Object*, objects of any type can be stored in an *ArrayList*. Java goes even further by providing "parameterized types," which were introduced in Subsection 7.3.4. There we saw that the *ArrayList* type can be parameterized, as in "*ArrayList<String>*", to limit the values that can be stored in the list to objects of a specified type. Parameterized types extend Java's basic philosophy of type-safe programming to generic programming.

The *ArrayList* class is just one of several standard classes that are used for generic programming in Java. We will spend the next few sections looking at these classes and how they are used, and we'll see that there are also generic methods and generic interfaces (see Subsection 5.7.1). All the classes and interfaces discussed in these sections are defined in the package `java.util`, and you will need an import statement at the beginning of your program to get access to them. (Before you start putting "`import java.util.*`" at the beginning of every program, you should know that some things in `java.util` have names that are the same as

things in other packages. For example, both `java.util.List` and `java.awt.List` exist, so it is often better to import the individual classes that you need.)

In the final section of this chapter, we will see that it is possible to define new generic classes, interfaces, and methods. Until then, we will stick to using the generics that are predefined in Java's standard library.

It is no easy task to design a library for generic programming. Java's solution has many nice features but is certainly not the only possible approach. It is almost certainly not the best, and has a few features that in my opinion can only be called bizarre, but in the context of the overall design of Java, it might be close to optimal. To get some perspective on generic programming in general, it might be useful to look very briefly at generic programming in two other languages.

10.1.1 Generic Programming in Smalltalk

Smalltalk was one of the very first object-oriented programming languages. It is still used today, although its use is not very common. It has not achieved anything like the popularity of Java or C++, but it is the source of many ideas used in these languages. In Smalltalk, essentially all programming is generic, because of two basic properties of the language.

First of all, variables in Smalltalk are typeless. A data value has a type, such as integer or string, but variables do not have types. Any variable can hold data of any type. Parameters are also typeless, so a subroutine can be applied to parameter values of any type. Similarly, a data structure can hold data values of any type. For example, once you've defined a binary tree data structure in SmallTalk, you can use it for binary trees of integers or strings or dates or data of any other type. There is simply no need to write new code for each data type.

Secondly, all data values are objects, and all operations on objects are defined by methods in a class. This is true even for types that are "primitive" in Java, such as integers. When the "`+`" operator is used to add two integers, the operation is performed by calling a method in the integer class. When you define a new class, you can define a "`+`" operator, and you will then be able to add objects belonging to that class by saying "`a + b`" just as if you were adding numbers. Now, suppose that you write a subroutine that uses the "`+`" operator to add up the items in a list. The subroutine can be applied to a list of integers, but it can also be applied, automatically, to any other data type for which "`+`" is defined. Similarly, a subroutine that uses the "`<`" operator to sort a list can be applied to lists containing any type of data for which "`<`" is defined. There is no need to write a different sorting subroutine for each type of data.

Put these two features together and you have a language where data structures and algorithms will work for any type of data for which they make sense, that is, for which the appropriate operations are defined. This is real generic programming. This might sound pretty good, and you might be asking yourself why all programming languages don't work this way. This type of freedom makes it easier to write programs, but unfortunately it makes it harder to write programs that are correct and robust (see Chapter 8). Once you have a data structure that can contain data of any type, it becomes hard to ensure that it only holds the type of data that you want it to hold. If you have a subroutine that can sort any type of data, it's hard to ensure that it will only be applied to data for which the "`<`" operator is defined. More particularly, there is no way for a **compiler** to ensure these things. The problem will only show up at run time when an attempt is made to apply some operation to a data type for which it is not defined, and the program will crash.

10.1.2 Generic Programming in C++

Unlike Smalltalk, C++ is a very strongly typed language, even more so than Java. Every variable has a type, and can only hold data values of that type. This means that the kind of generic programming that is used in Smalltalk is impossible in C++. Furthermore, C++ does not have anything corresponding to Java's *Object* class. That is, there is no class that is a superclass of all other classes. This means that C++ can't use Java's style of generic programming with non-parameterized generic types either. Nevertheless, C++ has a powerful and flexible system of generic programming. It is made possible by a language feature known as ***templates***. In C++, instead of writing a different sorting subroutine for each type of data, you can write a single subroutine template. The template is not a subroutine; it's more like a factory for making subroutines. We can look at an example, since the syntax of C++ is very similar to Java's:

```
template<class ItemType>
void sort( ItemType A[], int count ) {
      // Sort items in the array, A, into increasing order.
      // The items in positions 0, 1, 2, ..., (count-1) are sorted.
      // The algorithm that is used here is selection sort.
   for (int i = count-1; i > 0; i--) {
      int position_of_max = 0;
      for (int j = 1; j <= count ; j++)
         if ( A[j] > A[position_of_max] )
            position_of_max = j;
      ItemType temp = A[count];
      A[count] = A[position_of_max];
      A[position_of_max] = temp;
   }
}
```

This piece of code defines a subroutine template. If you remove the first line, "template<class ItemType>", and substitute the word "int" for the word "ItemType" in the rest of the template, you get a subroutine for sorting arrays of **ints**. (Even though it says "class ItemType", you can actually substitute any type for ItemType, including the primitive types.) If you substitute "string" for "ItemType", you get a subroutine for sorting arrays of strings. This is pretty much what the compiler does with the template. If your program says "`sort(list,10)`" where list is an array of **ints**, the compiler uses the template to generate a subroutine for sorting arrays of **ints**. If you say "`sort(cards,10)`" where cards is an array of objects of type *Card*, then the compiler generates a subroutine for sorting arrays of *Cards*. At least, it tries to. The template uses the "`>`" operator to compare values. If this operator is defined for values of type *Card*, then the compiler will successfully use the template to generate a subroutine for sorting cards. If "`>`" is not defined for *Cards*, then the compiler will fail—but this will happen at compile time, not, as in Smalltalk, at run time where it would make the program crash.

In addition to subroutine templates, C++ also has templates for making classes. If you write a template for a binary tree class, you can use it to generate classes for binary trees of **ints**, binary trees of strings, binary trees of dates, and so on—all from one template. The most recent version of C++ comes with a large number of pre-written templates called the ***Standard Template Library*** or STL. The STL is quite complex. Many people would say that it's much too complex. But it is also one of the most interesting features of C++.

10.1.3 Generic Programming in Java

Java's generic programming features have gone through several stages of development. The original version of Java had just a few generic data structure classes, such as *Vector*, that could hold values of type *Object*. Java version 1.2 introduced a much larger group of generics that followed the same basic model. These generic classes and interfaces as a group are known as the ***Java Collection Framework***. The *ArrayList* class is part of the Collection Framework. The original Collection Framework was closer in spirit to Smalltalk than it was to C++, since a data structure designed to hold *Objects* can be used with objects of any type. Unfortunately, as in Smalltalk, the result is a category of errors that show up only at run time, rather than at compile time. If a programmer assumes that all the items in a data structure are strings and tries to process those items as strings, a run-time error will occur if other types of data have inadvertently been added to the data structure. In Java, the error will most likely occur when the program retrieves an *Object* from the data structure and tries to type-cast it to type *String*. If the object is not actually of type *String*, the illegal type-cast will throw an error of type *ClassCastException*.

Java 5.0 introduced parameterized types, such as *ArrayList<String>*. This made it possible to create generic data structures that can be type-checked at compile time rather than at run time. With these data structures, type-casting is not necessary, so *ClassCastExceptions* are avoided. The compiler will detect any attempt to add an object of the wrong type to the data structure; it will report a syntax error and will refuse to compile the program. In Java 5.0, all of the classes and interfaces in the Collection Framework, and even some classes that are not part of that framework, have been parameterized. Java's parameterized classes are similar to template classes in C++ (although the implementation is very different), and their introduction moves Java's generic programming model closer to C++ and farther from Smalltalk. In this chapter, I will use the parameterized types almost exclusively, but you should remember that their use is not mandatory. It is still legal to use a parameterized class as a non-parameterized type, such as a plain *ArrayList*.

Note that there is a significant difference between parameterized classes in Java and template classes in C++. A template class in C++ is not really a class at all—it's a kind of factory for generating classes. Every time the template is used with a new type, a new compiled class is created. With a Java parameterized class, there is only one compiled class file. For example, there is only one compiled class file, `ArrayList.class`, for the parameterized class *ArrayList*. The parameterized types *ArrayList<String>* and *ArrayList<Integer>* both use the same compiled class file, as does the plain *ArrayList* type. The type parameter—*String* or *Integer*—just tells the compiler to limit the type of object that can be stored in the data structure. The type parameter has no effect at run time and is not even known at run time. The type information is said to be "erased" at run time. This ***type erasure*** introduces a certain amount of weirdness. For example, you can't test "`if (list instanceof ArrayList<String>)`" because the `instanceof` operator is evaluated at run time, and at run time only the plain *ArrayList* exists. Even worse, you can't create an array that has base type *ArrayList<String>* by using the `new` operator, as in "`new ArrayList<String>[N]`". This is because the `new` operator is evaluated at run time, and at run time there is no such thing as "`ArrayList<String>`"; only the non-parameterized type *ArrayList* exists at run time.

Fortunately, most programmers don't have to deal with such problems, since they turn up only in fairly advanced programming. Most people who use the Java Collection Framework will not encounter them, and they will get the benefits of type-safe generic programming with little difficulty.

10.1.4 The Java Collection Framework

Java's generic data structures can be divided into two categories: ***collections*** and ***maps***. A collection is more or less what it sounds like: a collection of objects. A map associates objects in one set with objects in another set in the way that a dictionary associates definitions with words or a phone book associates phone numbers with names. A map is similar to what I called an "association list" in Subsection 7.4.2. In Java, collections and maps are represented by the parameterized interfaces *Collection<T>* and *Map<T,S>*. Here, "T" and "S" stand for any type except for the primitive types. *Map<T,S>* is the first example we have seen where there are two type parameters, *T* and *S*; we will not deal further with this possibility until we look at maps more closely in Section 10.3. In this section and the next, we look at collections only.

There are two types of collections: ***lists*** and ***sets***. A list is a collection in which the objects are arranged in a linear sequence. A list has a first item, a second item, and so on. For any item in the list, except the last, there is an item that directly follows it. The defining property of a set is that no object can occur more than once in a set; the elements of a set are not necessarily thought of as being in any particular order. The ideas of lists and sets are represented as parameterized interfaces *List<T>* and *Set<T>*. These are sub-interfaces of *Collection<T>*. That is, any object that implements the interface *List<T>* or *Set<T>* automatically implements *Collection<T>* as well. The interface *Collection<T>* specifies general operations that can be applied to any collection at all. *List<T>* and *Set<T>* add additional operations that are appropriate for lists and sets respectively.

Of course, any actual object that is a collection, list, or set must belong to a concrete class that implements the corresponding interface. For example, the class *ArrayList<T>* implements the interface *List<T>* and therefore also implements *Collection<T>*. This means that all the methods that are defined in the list and collection interfaces can be used with, for example, an *ArrayList<String>* object. We will look at various classes that implement the list and set interfaces in the next section. But before we do that, we'll look briefly at some of the general operations that are available for all collections.

* * *

The interface *Collection<T>* specifies methods for performing some basic operations on any collection of objects. Since "collection" is a very general concept, operations that can be applied to all collections are also very general. They are generic operations in the sense that they can be applied to various types of collections containing various types of objects. Suppose that `coll` is an object that implements the interface *Collection<T>* (for some specific non-primitive type *T*). Then the following operations, which are specified in the interface *Collection<T>*, are defined for `coll`:

- `coll.size()` — returns an **int** that gives the number of objects in the collection.
- `coll.isEmpty()` — returns a **boolean** value which is `true` if the size of the collection is 0.
- `coll.clear()` — removes all objects from the collection.
- `coll.add(tobject)` — adds `tobject` to the collection. The parameter must be of type *T*; if not, a syntax error occurs at compile time. This method returns a **boolean** value which tells you whether the operation actually modified the collection. For example, adding an object to a `Set` has no effect if that object was already in the set.
- `coll.contains(object)` — returns a **boolean** value that is true if `object` is in the collection. Note that `object` is **not** required to be of type *T*, since it makes sense to check whether `object` is in the collection, no matter what type `object` has. (For testing

equality, `null` is considered to be equal to itself. The criterion for testing non-null objects for equality can differ from one kind of collection to another; see Subsection 10.1.6, below.)

- `coll.remove(object)` — removes `object` from the collection, if it occurs in the collection, and returns a **boolean** value that tells you whether the object was found. Again, `object` is not required to be of type *T*.
- `coll.containsAll(coll2)` — returns a **boolean** value that is true if every `object` in `coll2` is also in `coll`. The parameter can be any collection.
- `coll.addAll(coll2)` — adds all the objects in `coll2` to `coll`. The parameter, `coll2`, can be any collection of type *Collection<T>*. However, it can also be more general. For example, if *T* is a class and *S* is a sub-class of *T*, then `coll2` can be of type *Collection<S>*. This makes sense because any object of type *S* is automatically of type *T* and so can legally be added to `coll`.
- `coll.removeAll(coll2)` — removes every `object` from `coll` that also occurs in the collection `coll2`. `coll2` can be any collection.
- `coll.retainAll(coll2)` — removes every `object` from `coll` that **does not occur** in the collection `coll2`. It "retains" only the objects that do occur in `coll2`. `coll2` can be any collection.
- `coll.toArray()` — returns an array of type Object[] that contains all the items in the collection. Note that the return type is Object[], not T[]! However, there is another version of this method that takes an array of type T[] as a parameter: the method `coll.toArray(tarray)` returns an array of type T[] containing all the items in the collection. If the array parameter `tarray` is large enough to hold the entire collection, then the items are stored in `tarray` and `tarray` is also the return value of the collection. If `tarray` is not large enough, then a new array is created to hold the items; in that case `tarray` serves only to specify the type of the array. For example, `coll.toArray(new String[0])` can be used if `coll` is a collection of *Strings* and will return a new array of type String[].

Since these methods are part of the *Collection<T>* interface, they must be defined for every object that implements that interface. There is a problem with this, however. For example, the size of some collections cannot be changed after they are created. Methods that add or remove objects don't make sense for these collections. While it is still legal to call the methods, an exception will be thrown when the call is evaluated at run time. The type of the exception is *UnsupportedOperationException*. Furthermore, since *Collection<T>* is only an interface, not a concrete class, the actual implementation of the method is left to the classes that implement the interface. This means that the semantics of the methods, as described above, are not guaranteed to be valid for all collection objects; they are valid, however, for classes in the Java Collection Framework.

There is also the question of efficiency. Even when an operation is defined for several types of collections, it might not be equally efficient in all cases. Even a method as simple as `size()` can vary greatly in efficiency. For some collections, computing the `size()` might involve counting the items in the collection. The number of steps in this process is equal to the number of items. Other collections might have instance variables to keep track of the size, so evaluating `size()` just means returning the value of a variable. In this case, the computation takes only one step, no matter how many items there are. When working with collections, it's good to have some idea of how efficient operations are and to choose a collection for which the operations that you need can be implemented most efficiently. We'll see specific examples of this in the next two sections.

10.1.5 Iterators and for-each Loops

The interface *Collection<T>* defines a few basic generic algorithms, but suppose you want to write your own generic algorithms. Suppose, for example, you want to do something as simple as printing out every item in a collection. To do this in a generic way, you need some way of going through an arbitrary collection, accessing each item in turn. We have seen how to do this for specific data structures: For an array, you can use a `for` loop to iterate through all the array indices. For a linked list, you can use a while loop in which you advance a pointer along the list. For a binary tree, you can use a recursive subroutine to do an inorder traversal. Collections can be represented in any of these forms and many others besides. With such a variety of traversal mechanisms, how can we even hope to come up with a single generic method that will work for collections that are stored in wildly different forms? This problem is solved by ***iterators***. An iterator is an object that can be used to traverse a collection. Different types of collections have iterators that are implemented in different ways, but all iterators are **used** in the same way. An algorithm that uses an iterator to traverse a collection is generic, because the same technique can be applied to any type of collection. Iterators can seem rather strange to someone who is encountering generic programming for the first time, but you should understand that they solve a difficult problem in an elegant way.

The interface *Collection<T>* defines a method that can be used to obtain an iterator for any collection. If `coll` is a collection, then `coll.iterator()` returns an iterator that can be used to traverse the collection. You should think of the iterator as a kind of generalized pointer that starts at the beginning of the collection and can move along the collection from one item to the next. Iterators are defined by a parameterized interface named *Iterator<T>*. If `coll` implements the interface *Collection<T>* for some specific type *T*, then `coll.iterator()` returns an iterator of type *Iterator<T>*, with the same type *T* as its type parameter. The interface *Iterator<T>* defines just three methods. If `iter` refers to an object that implements *Iterator<T>*, then we have:

- `iter.next()` — returns the next item, and advances the iterator. The return value is of type *T*. This method lets you look at one of the items in the collection. Note that there is no way to look at an item without advancing the iterator past that item. If this method is called when no items remain, it will throw a *NoSuchElementException*.
- `iter.hasNext()` — returns a **boolean** value telling you whether there are more items to be processed. In general, you should test this before calling `iter.next()`.
- `iter.remove()` — if you call this after calling `iter.next()`, it will remove the item that you just saw from the collection. Note that this method has **no parameter**. It removes the item that was most recently returned by `iter.next()`. This might produce an *UnsupportedOperationException*, if the collection does not support removal of items.

Using iterators, we can write code for printing all the items in **any** collection. Suppose, for example, that `coll` is of type *Collection<String>*. In that case, the value returned by `coll.iterator()` is of type *Iterator<String>*, and we can say:

```
Iterator<String> iter;          // Declare the iterator variable.
iter = coll.iterator();         // Get an iterator for the collection.
while ( iter.hasNext() ) {
   String item = iter.next();   // Get the next item.
   System.out.println(item);
}
```

The same general form will work for other types of processing. For example, the following code will remove all `null` values from any collection of type *Collection<JButton>* (as long as that collection supports removal of values):

```
Iterator<JButton> iter = coll.iterator():
while ( iter.hasNext() ) {
    JButton item = iter.next();
    if (item == null)
       iter.remove();
}
```

(Note, by the way, that when *Collection<T>*, *Iterator<T>*, or any other parameterized type is used in actual code, they are always used with actual types such as *String* or *JButton* in place of the "formal type parameter" *T*. An iterator of type *Iterator<String>* is used to iterate through a collection of *Strings*; an iterator of type *Iterator<JButton>* is used to iterate through a collection of *JButtons*; and so on.)

An iterator is often used to apply the same operation to all the elements in a collection. In many cases, it's possible to avoid the use of iterators for this purpose by using a for-each loop. The for-each loop was discussed in Subsection 3.4.4 for use with enumerated types and in Subsection 7.2.2 for use with arrays. A for-each loop can also be used to iterate through any collection. For a collection `coll` of type *Collection<T>*, a for-each loop takes the form:

```
for ( T x : coll ) { // "for each object x, of type T, in coll"
   //  process x
}
```

Here, `x` is the loop control variable. Each object in `coll` will be assigned to `x` in turn, and the body of the loop will be executed for each object. Since objects in `coll` are of type *T*, `x` is declared to be of type *T*. For example, if `namelist` is of type *Collection<String>*, we can print out all the names in the collection with:

```
for ( String name : namelist ) {
   System.out.println( name );
}
```

This for-each loop could, of course, be written as a `while` loop using an iterator, but the for-each loop is much easier to follow.

10.1.6 Equality and Comparison

There are several methods in the *Collection* interface that test objects for equality. For example, the methods `coll.contains(object)` and `coll.remove(object)` look for an item in the collection that is equal to `object`. However, equality is not such a simple matter. The obvious technique for testing equality—using the `==` operator—does not usually give a reasonable answer when applied to objects. The `==` operator tests whether two objects are identical in the sense that they share the same location in memory. Usually, however, we want to consider two objects to be equal if they represent the same value, which is a very different thing. Two values of type *String* should be considered equal if they contain the same sequence of characters. The question of whether those characters are stored in the same location in memory is irrelevant. Two values of type *Date* should be considered equal if they represent the same time.

The *Object* class defines the **boolean**-valued method `equals(Object)` for testing whether one object is equal to another. This method is used by many, but not by all, collection classes for deciding whether two objects are to be considered the same. In the *Object* class,

`obj1.equals(obj2)` is defined to be the same as `obj1 == obj2`. However, for most sub-classes of `Object`, this definition is not reasonable, and it should be overridden. The *String* class, for example, overrides `equals()` so that for a *String* `str`, `str.equals(obj)` if `obj` is also a *String* and `obj` contains the same sequence of characters as `str`.

If you write your own class, you might want to define an `equals()` method in that class to get the correct behavior when objects are tested for equality. For example, a *Card* class that will work correctly when used in collections could be defined as:

```
public class Card {  // Class to represent playing cards.

   int suit;  // Number from 0 to 3 that codes for the suit --
              // spades, diamonds, clubs or hearts.
   int value; // Number from 1 to 13 that represents the value.

   public boolean equals(Object obj) {
       try {
          Card other = (Card)obj;  // Type-cast obj to a Card.
          if (suit == other.suit && value == other.value) {
                // The other card has the same suit and value as
                // this card, so they should be considered equal.
             return true;
          }
          else
             return false;
       }
       catch (Exception e) {
              // This will catch the NullPointerException that occurs if obj
              // is null and the ClassCastException that occurs if obj is
              // not of type Card.  In these cases, obj is not equal to
              // this Card, so return false.
           return false;
       }
    }

    .
    . // other methods and constructors
    .
}
```

Without the `equals()` method in this class, methods such as `contains()` and `remove()` in the interface *Collection<Card>* will not work as expected.

A similar concern arises when items in a collection are sorted. Sorting refers to arranging a sequence of items in ascending order, according to some criterion. The problem is that there is no natural notion of ascending order for arbitrary objects. Before objects can be sorted, some method must be defined for comparing them. Objects that are meant to be compared should implement the interface `java.lang.Comparable`. In fact, *Comparable* is defined as a parameterized interface, *Comparable<T>*, which represents the ability to be compared to an object of type *T*. The interface *Comparable<T>* defines one method:

```
public int compareTo( T obj )
```

The value returned by `obj1.compareTo(obj2)` should be negative if and only if `obj1` comes before `obj2`, when the objects are arranged in ascending order. It should be positive if and only if `obj1` comes after `obj2`. A return value of zero means that the objects are considered

to be the same for the purposes of this comparison. This does not necessarily mean that the objects are equal in the sense that `obj1.equals(obj2)` is true. For example, if the objects are of type *Address*, representing mailing addresses, it might be useful to sort the objects by zip code. Two *Addresses* are considered the same for the purposes of the sort if they have the same zip code—but clearly that would not mean that they are the same address.

The *String* class implements the interface *Comparable*<*String*> and defines `compareTo` in a reasonable way. In this case, the return value of `compareTo` is zero if and only if the two strings that are being compared are equal. (It is generally a good idea for the `compareTo` method in classes that implement *Comparable* to have the analogous property.) If you define your own class and want to be able to sort objects belonging to that class, you should do the same. For example:

```
/**
 * Represents a full name consisting of a first name and a last name.
 */
public class FullName implements Comparable<FullName> {

   private String firstName, lastName;  // Non-null first and last names.

   public FullName(String first, String last) {  // Constructor.
      if (first == null || last == null)
         throw new IllegalArgumentException("Names must be non-null.");
      firstName = first;
      lastName = last;
   }

   public boolean equals(Object obj) {
      try {
         FullName other = (FullName)obj;  // Type-cast obj to type FullName
         return firstName.equals(other.firstName)
                                 && lastName.equals(other.lastName);
      }
      catch (Exception e) {
         return false;  // if obj is null or is not of type FullName
      }
   }

   public int compareTo( FullName other ) {
      if ( lastName.compareTo(other.lastName) < 0 ) {
             // If lastName comes before the last name of
             // the other object, then this FullName comes
             // before the other FullName.  Return a negative
             // value to indicate this.
         return -1;
      }
      else if ( lastName.compareTo(other.lastName) > 0 ) {
             // If lastName comes after the last name of
             // the other object, then this FullName comes
             // after the other FullName.  Return a positive
             // value to indicate this.
         return 1;
      }
      else {
             // Last names are the same, so base the comparison on
```

```
                // the first names, using compareTo from class String.
            return firstName.compareTo(other.firstName);
        }
    }

    .
    . // other methods
    .
}
```

(I find it a little odd that the class here is declared as "`class FullName implements Comparable<FullName>`", with "FullName" repeated as a type parameter in the name of the interface. However, it does make sense. It means that we are going to compare objects that belong to the class *FullName* to other objects **of the same type**. Even though this is the only reasonable thing to do, that fact is not obvious to the Java compiler—and the type parameter in *Comparable*<*FullName*> is there for the compiler.)

There is another way to allow for comparison of objects in Java, and that is to provide a separate object that is capable of making the comparison. The object must implement the interface *Comparator*<*T*>, where *T* is the type of the objects that are to be compared. The interface *Comparator*<*T*> defines the method:

```
public int compare( T obj1, T obj2 )
```

This method compares two objects of type *T* and returns a value that is negative, or positive, or zero, depending on whether `obj1` comes before `obj2`, or comes after `obj2`, or is considered to be the same as `obj2` for the purposes of this comparison. Comparators are useful for comparing objects that do not implement the *Comparable* interface and for defining several different orderings on the same collection of objects.

In the next two sections, we'll see how *Comparable* and *Comparator* are used in the context of collections and maps.

10.1.7 Generics and Wrapper Classes

As noted above, Java's generic programming does not apply to the primitive types, since generic data structures can only hold objects, while values of primitive type are not objects. However, the "wrapper classes" that were introduced in Subsection 5.3.2 make it possible to get around this restriction to a great extent.

Recall that each primitive type has an associated wrapper class: class *Integer* for type **int**, class *Boolean* for type **boolean**, class *Character* for type **char**, and so on.

An object of type *Integer* contains a value of type **int**. The object serves as a "wrapper" for the primitive type value, which allows it to be used in contexts where objects are required, such as in generic data structures. For example, a list of *Integers* can be stored in a variable of type *ArrayList*<*Integer*>, and interfaces such as *Collection*<*Integer*> and *Set*<*Integer*> are defined. Furthermore, class *Integer* defines `equals()`, `compareTo()`, and `toString()` methods that do what you would expect (that is, that compare and write out the corresponding primitive type values in the usual way). Similar remarks apply for all the wrapper classes.

Recall also that Java does automatic conversions between a primitive type and the corresponding wrapper type. (These conversions, which are called autoboxing and unboxing, were also introduced in Subsection 5.3.2.) This means that once you have created a generic data structure to hold objects belonging to one of the wrapper classes, you can use the data structure

pretty much as if it actually contained primitive type values. For example, if `numbers` is a variable of type *Collection<Integer>*, it is legal to call `numbers.add(17)` or `numbers.remove(42)`. You can't literally add the primitive type value 17 to `numbers`, but Java will automatically convert the 17 to the corresponding wrapper object, `new Integer(17)`, and the wrapper object will be added to the collection. (The creation of the object does add some time and memory overhead to the operation, and you should keep that in mind in situations where efficiency is important. An array of **int** is more efficient than an *ArrayList<Integer>*.)

10.2 Lists and Sets

In the previous section, we looked at the general properties of collection classes in Java. In this section, we look at some specific collection classes and how to use them. These classes can be divided into two categories: lists and sets. A list consists of a sequence of items arranged in a linear order. A list has a definite order, but is not necessarily sorted into ascending order. A set is a collection that has no duplicate entries. The elements of a set might or might not be arranged into some definite order.

10.2.1 ArrayList and LinkedList

There are two obvious ways to represent a list: as a dynamic array and as a linked list. We've encountered these already in Section 7.3 and Section 9.2. Both of these options are available in generic form as the collection classes `java.util.ArrayList` and `java.util.LinkedList`. These classes are part of the Java Collection Framework. Each implements the interface *List<T>*, and therefore the interface *Collection<T>*. An object of type *ArrayList<T>* represents an ordered sequence of objects of type *T*, stored in an array that will grow in size whenever necessary as new items are added. An object of type *LinkedList<T>* also represents an ordered sequence of objects of type *T*, but the objects are stored in nodes that are linked together with pointers.

Both list classes support the basic list operations that are defined in the interface *List<T>*, and an abstract data type is defined by its operations, not by its representation. So why two classes? Why not a single List class with a single representation? The problem is that there **is** no single representation of lists for which all list operations are efficient. For some operations, linked lists are more efficient than arrays. For others, arrays are more efficient. In a particular application of lists, it's likely that only a few operations will be used frequently. You want to choose the representation for which the frequently used operations will be as efficient as possible.

Broadly speaking, the *LinkedList* class is more efficient in applications where items will often be added or removed at the beginning of the list or in the middle of the list. In an array, these operations require moving a large number of items up or down one position in the array, to make a space for a new item or to fill in the hole left by the removal of an item. In terms of asymptotic analysis (Section 8.5), adding an element at the beginning or in the middle of an array has run time $\Theta(n)$, where n is the number of items in the array. In a linked list, nodes can be added or removed at any position by changing a few pointer values, an operation that has run time $\Theta(1)$. That is, the operation takes only some constant amount of time, independent of how many items are in the list.

On the other hand, the *ArrayList* class is more efficient when ***random access*** to items is required. Random access means accessing the k-th item in the list, for any integer k. Random access is used when you get or change the value stored at a specified position in the list. This is

trivial for an array, with run time $\Theta(1)$. But for a linked list it means starting at the beginning of the list and moving from node to node along the list for k steps, an operation that has run time $\Theta(k)$.

Operations that can be done efficiently for both types of lists include sorting and adding an item at the end of the list.

All lists implement the methods from interface *Collection<T>* that were discussed in Subsection 10.1.4. These methods include `size()`, `isEmpty()`, `add(T)`, `remove(Object)`, and `clear()`. The `add(T)` method adds the object at the end of the list. The `remove(Object)` method involves first finding the object, which is not very efficient for any list since it involves going through the items in the list from beginning to end until the object is found. The interface *List<T>* adds some methods for accessing list items according to their numerical positions in the list. Suppose that `list` is an object of type *List<T>*. Then we have the methods:

- `list.get(index)` — returns the object of type *T* that is at position `index` in the list, where `index` is an integer. Items are numbered 0, 1, 2, ..., `list.size()-1`. The parameter must be in this range, or an *IndexOutOfBoundsException* is thrown.
- `list.set(index,obj)` — stores the object `obj` at position number `index` in the list, replacing the object that was there previously. The object `obj` must be of type *T*. This does not change the number of elements in the list or move any of the other elements.
- `list.add(index,obj)` — inserts an object `obj` into the list at position number `index`, where `obj` must be of type *T*. The number of items in the list increases by one, and items that come after position `index` move down one position to make room for the new item. The value of `index` must be in the range 0 to `list.size()`, inclusive. If `index` is equal to `list.size()`, then `obj` is added at the end of the list.
- `list.remove(index)` — removes the object at position number `index`, and returns that object as the return value of the method. Items after this position move up one space in the list to fill the hole, and the size of the list decreases by one. The value of `index` must be in the range 0 to `list.size()-1`
- `list.indexOf(obj)` — returns an **int** that gives the position of `obj` in the list, if it occurs. If it does not occur, the return value is `-1`. The object `obj` can be of any type, not just of type *T*. If `obj` occurs more than once in the list, the index of the first occurrence is returned.

These methods are defined both in class *ArrayList<T>* and in class *LinkedList<T>*, although some of them—`get` and `set`—are only efficient for *ArrayLists*. The class *LinkedList<T>* adds a few additional methods, which are not defined for an *ArrayList*. If `linkedlist` is an object of type *LinkedList<T>*, then we have

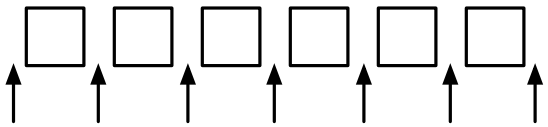
- `linkedlist.getFirst()` — returns the object of type *T* that is the first item in the list. The list is not modified. If the list is empty when the method is called, an exception of type *NoSuchElementException* is thrown (the same is true for the next three methods as well).
- `linkedlist.getLast()` — returns the object of type *T* that is the last item in the list. The list is not modified.
- `linkedlist.removeFirst()` — removes the first item from the list, and returns that object of type *T* as its return value.
- `linkedlist.removeLast()` — removes the last item from the list, and returns that object of type *T* as its return value.

- `linkedlist.addFirst(obj)` — adds the `obj`, which must be of type *T*, to the beginning of the list.
- `linkedlist.addLast(obj)` — adds the object `obj`, which must be of type *T*, to the end of the list. (This is exactly the same as `linkedlist.add(obj)` but is defined to keep the naming consistent.)

These methods are apparently defined to make it easy to use a *LinkedList* as if it were a stack or a queue. (See Section 9.3.) For example, we can use a *LinkedList* as a queue by adding items onto one end of the list (using the `addLast()` method) and removing them from the other end (using the `removeFirst()` method).

If `list` is an object of type *List<T>*, then the method `list.iterator()`, defined in the interface *Collection<T>*, returns an *Iterator* that can be used to traverse the list from beginning to end. However, for *Lists*, there is a special type of *Iterator*, called a *ListIterator*, which offers additional capabilities. *ListIterator<T>* is an interface that extends the interface *Iterator<T>*. The method `list.listIterator()` returns an object of type *ListIterator<T>*.

A *ListIterator* has the usual *Iterator* methods, `hasNext()`, `next()`, and `remove()`, but it also has methods `hasPrevious()`, `previous()`, and `add(obj)` that make it possible to move backwards in the list and to add an item at the current position of the iterator. To understand how these work, it's best to think of an iterator as pointing to a position **between** two list elements, or at the beginning or end of the list. In this diagram, the items in a list are represented by squares, and arrows indicate the possible positions of an iterator:



If `iter` is of type *ListIterator<T>*, then `iter.next()` moves the iterator one space to the right along the list and returns the item that the iterator passes as it moves. The method `iter.previous()` moves the iterator one space to the left along the list and returns the item that it passes. The method `iter.remove()` removes an item from the list; the item that is removed is the item that the iterator passed most recently in a call to either `iter.next()` or `iter.previous()`. There is also a method `iter.add(obj)` that adds the specified object to the list at the current position of the iterator (where `obj` must be of type *T*). This can be between two existing items or at the beginning of the list or at the end of the list.

(By the way, the lists that are used in class `LinkedList<T>` are ***doubly linked lists***. That is, each node in the list contains two pointers—one to the next node in the list and one to the previous node. This makes it possible to efficiently implement both the `next()` and `previous()` methods of a `ListIterator`. Also, to make the `addLast()` and `getLast()` methods of a *LinkedList* efficient, the class *LinkedList<T>* includes an instance variable that points to the last node in the list.)

As an example of using a *ListIterator*, suppose that we want to maintain a list of items that is always sorted into increasing order. When adding an item to the list, we can use a *ListIterator* to find the position in the list where the item should be added. Once the position has been found, we use the same list iterator to place the item in that position. The idea is to start at the beginning of the list and to move the iterator forward past all the items that are smaller than the item that is being inserted. At that point, the iterator's `add()` method can be used to insert the item. To be more definite, suppose that `stringList` is a variable of type *List<String>*.

Assume that that the strings that are already in the list are stored in ascending order and that `newItem` is a string that we would like to insert into the list. The following code will place `newItem` in the list in its correct position, so that the modified list is still in ascending order:

```
ListIterator<String> iter = stringList.listIterator();

// Move the iterator so that it points to the position where
// newItem should be inserted into the list.  If newItem is
// bigger than all the items in the list, then the while loop
// will end when iter.hasNext() becomes false, that is, when
// the iterator has reached the end of the list.

while (iter.hasNext()) {
   String item = iter.next();
   if (newItem.compareTo(item) <= 0) {
         // newItem should come BEFORE item in the list.
         // Move the iterator back one space so that
         // it points to the correct insertion point,
         // and end the loop.
      iter.previous();
      break;
   }
}

iter.add(newItem);
```

Here, `stringList` might be of type *ArrayList<String>* or of type *LinkedList<String>*. The algorithm that is used to insert `newItem` into the list will be about equally efficient for both types of lists, and it will even work for other classes that implement the interface *List<String>*. You would probably find it easier to design an insertion algorithm that uses array-like indexing with the methods `get(index)` and `add(index,obj)`. However, that algorithm would be horribly inefficient for *LinkedLists* because random access is so inefficient for linked lists. (By the way, the insertion algorithm works when the list is empty. It might be useful for you to think about why this is true.)

10.2.2 Sorting

Sorting a list is a fairly common operation, and there should really be a sorting method in the *List* interface. There is not, presumably because it only makes sense to sort lists of certain types of objects, but methods for sorting lists are available as `static` methods in the class `java.util.Collections`. This class contains a variety of static utility methods for working with collections. The methods are generic; that is, they will work for collections of objects of various types. Suppose that `list` is of type *List<T>*. The command

```
Collections.sort(list);
```

can be used to sort the list into ascending order. The items in the list should implement the interface *Comparable<T>* (see Subsection 10.1.6). The method `Collections.sort()` will work, for example, for lists of *String* and for lists of any of the wrapper classes such as *Integer* and *Double*. There is also a sorting method that takes a *Comparator* as its second argument:

```
Collections.sort(list,comparator);
```

In this method, the `comparator` will be used to compare the items in the list. As mentioned in the previous section, a *Comparator* is an object that defines a `compare()` method that can be used to compare two objects. We'll see an example of using a *Comparator* in Section 10.4.

The sorting method that is used by `Collections.sort()` is the so-called "merge sort" algorithm, which has both worst-case and average-case run times that are $\Theta(n*\log(n))$ for a list of size n. Although the average run time for MergeSort is a little slower than that of QuickSort, its worst-case performance is much better than QuickSort's. (QuickSort was covered in Subsection 9.1.3.) MergeSort also has a nice property called "stability" that we will encounter at the end of Subsection 10.4.3.

The *Collections* class has at least two other useful methods for modifying lists. `Collections.shuffle(list)` will rearrange the elements of the list into a random order. `Collections.reverse(list)` will reverse the order of the elements, so that the last element is moved to the beginning of the list, the next-to-last element to the second position, and so on.

Since an efficient sorting method is provided for *Lists*, there is no need to write one yourself. You might be wondering whether there is an equally convenient method for standard arrays. The answer is yes. Array-sorting methods are available as static methods in the class `java.util.Arrays`. The statement

```
Arrays.sort(A);
```

will sort an array, `A`, provided either that the base type of `A` is one of the primitive types (except **boolean**) or that `A` is an array of *Objects* that implement the `Comparable` interface. You can also sort part of an array. This is important since arrays are often only "partially filled." The command:

```
Arrays.sort(A,fromIndex,toIndex);
```

sorts the elements `A[fromIndex]`, `A[fromIndex+1]`, ..., `A[toIndex-1]` into ascending order. You can use `Arrays.sort(A,0,N)` to sort a partially filled array which has items in the first `N` positions.

Java does not support generic programming for primitive types. In order to implement the command `Arrays.sort(A)`, the `Arrays` class contains eight methods: one method for arrays of *Objects* and one method for each of the primitive types **byte**, **short**, **int**, **long**, **float**, **double**, and **char**.

10.2.3 TreeSet and HashSet

A set is a collection of objects in which no object occurs more than once. Sets implement all the methods in the interface *Collection<T>*, but do so in a way that ensures that no element occurs twice in the set. For example, if `set` is an object of type *Set<T>*, then `set.add(obj)` will have no effect on the set if `obj` is already an element of the set. Java has two classes that implement the interface *Set<T>*: `java.util.TreeSet` and `java.util.HashSet`.

In addition to being a *Set*, a *TreeSet* has the property that the elements of the set are arranged into ascending sorted order. An *Iterator* (or a for-each loop) for a `TreeSet` will always visit the elements of the set in ascending order.

A *TreeSet* cannot hold arbitrary objects, since there must be a way to determine the sorted order of the objects it contains. Ordinarily, this means that the objects in a set of type *TreeSet<T>* should implement the interface *Comparable<T>* and that `obj1.compareTo(obj2)` should be defined in a reasonable way for any two objects `obj1` and `obj2` in the set. Alternatively, an object of type *Comparator<T>* can be provided as a parameter to the constructor

when the *TreeSet* is created. In that case, the `compareTo()` method of the *Comparator* will be used to compare objects that are added to the set.

A *TreeSet* does not use the `equals()` method to test whether two objects are the same. Instead, it uses the `compareTo()` method. This can be a problem. Recall from Subsection 10.1.6 that `compareTo()` can consider two objects to be the same for the purpose of the comparison even though the objects are not equal. For a *TreeSet*, this means that only **one** of those objects can be in the set. For example, if the *TreeSet* contains mailing addresses and if the `compareTo()` method for addresses just compares their zip codes, then the set can contain only one address in each zip code. Clearly, this is not right! But that only means that you have to be aware of the semantics of *TreeSets*, and you need to make sure that `compareTo()` is defined in a reasonable way for objects that you put into a *TreeSet*. This will be true, by the way, for *Strings*, *Integers*, and many other built-in types, since the `compareTo()` method for these types considers two objects to be the same only if they are actually equal.

In the implementation of a *TreeSet*, the elements are stored in something similar to a binary sort tree. (See Subsection 9.4.2.) However, the data structure that is used is **balanced** in the sense that all the leaves of the tree are at about the same distance from the root of the tree. This ensures that all the basic operations—inserting, deleting, and searching—are efficient, with worst-case run time $\Theta(\log(n))$, where n is the number of items in the set.

The fact that a *TreeSet* sorts its elements and removes duplicates makes it very useful in some applications. Exercise 7.6 asked you to write a program that would read a file and output an alphabetical list of all the words that occurred in the file, with duplicates removed. The words were to be stored in an *ArrayList*, so it was up to you to make sure that the list was sorted and contained no duplicates. The same task can be programmed much more easily using a *TreeSet* instead of a list. A *TreeSet* automatically eliminates duplicates, and an iterator for the set will automatically visit the items in the set in sorted order. An algorithm for the program, using a *TreeSet*, would be:

```
TreeSet<String> words = new TreeSet<String>();

while there is more data in the input file:
   Let word = the next word from the file
   Convert word to lower case
   words.add(word)   // Adds the word only if not already present.

for ( String w : words ) // for each String w in words
   Output w
```

If you would like to see a complete, working program, you can find it in the file *WordListWithTreeSet.java*.

As another example, suppose that `coll` is any `Collection` of *Strings*. (This would also work for any other type for which `compareTo()` is properly defined.) We can use a *TreeSet* to sort the items of `coll` and remove the duplicates simply by saying:

```
TreeSet<String> set = new TreeSet<String>();
set.addAll(coll);
```

The second statement adds all the elements of the collection to the set. Since it's a *Set*, duplicates are ignored. Since it's a *TreeSet*, the elements of the set are sorted. If you would like to have the data in some other type of data structure, it's easy to copy the data from the set. For example, to place the answer in an *ArrayList*, you could say:

```
TreeSet<String> set = new TreeSet<String>();
set.addAll(coll);
```

```
ArrayList<String> list = new ArrayList<String>();
list.addAll(set);
```

Now, in fact, every one of Java's collection classes has a constructor that takes a *Collection* as an argument. All the items in that *Collection* are added to the new collection when it is created. So, if `coll` is of type *Collection*<*String*>, then "`new TreeSet<String>(coll)`" creates a *TreeSet* that contains the same elements as `coll`, but with duplicates removed and in sorted order. This means that we can abbreviate the four lines in the above example to the single command:

```
ArrayList<String> list = new ArrayList<String>( new TreeSet<String>(coll) );
```

This makes a sorted list of the elements of `coll` with no duplicates. Although the repeated type parameter, "`<String>`", makes it a bit ugly to look at, this is still a nice example of the power of generic programming. (It seems, by the way, there is no equally easy way to get a sorted list **with** duplicates. To do this, we would need something like a *TreeSet* that allows duplicates. The C++ programming language has such a thing and refers to it as a ***multiset***. The Smalltalk language has something similar and calls it a ***bag***. Java, for the time being at least, lacks this data type.)

* * *

A *HashSet* stores its elements in a ***hash table***, a type of data structure that I will discuss in the next section. The operations of finding, adding, and removing elements are implemented very efficiently in hash tables, even more so than for *TreeSets*. The elements of a *HashSet* are not stored in any particular order, and so do not need to implement the *Comparable* interface. (They do, however, need to define a proper "hash code," as we'll see in the next section.)

The `equals()` method is used to determine whether two objects in a *HashSet* are to be considered the same. An *Iterator* for a *HashSet* will visit its elements in what seems to be a completely arbitrary order, and it's possible for the order to change completely when a new element is added. Use a *HashSet* instead of a *TreeSet* when the elements it contains are not comparable, or when the order is not important, or when the small advantage in efficiency is important.

* * *

A note about the mathematics of sets: In mathematical set theory, the items in a set are called ***members*** or ***elements*** of that set. Important operations include adding an element to a set, removing an element from a set, and testing whether a given entity is an element of a set. Operations that can be performed on two sets include ***union***, ***intersection***, and ***set difference***. All these operations are defined in Java for objects of type *Set*, but with different names. Suppose that `A` and `B` are *Sets*. Then:

- `A.add(x)` **adds** the element `x` to the set `A`.
- `A.remove(x)` **removes** the element `x` from the set `A`.
- `A.contains(x)` **tests** whether `x` is an element of the set `A`.
- `A.addAll(B)` computes the **union** of `A` and `B`.
- `A.retainAll(B)` computes the **intersection** of `A` and `B`.
- `A.removeAll(B)` computes the **set difference**, `A - B`.

There are of course, differences between mathematical sets and sets in Java. Most important, perhaps, sets in Java must be finite, while in mathematics, most of the fun in set theory comes from working with infinity. In mathematics, a set can contain arbitrary elements, while in Java,

a set of type *Set<T>* can only contain elements of type *T*. The operation `A.addAll(B)` acts by modifying the value of `A`, while in mathematics the operation `A` union `B` computes a new set, without changing the value of `A` or `B`. See Exercise 10.2 for an example of mathematical set operations in Java.

10.2.4 EnumSet

Enumerated types (or "enums") were introduced in Subsection 2.3.3. Suppose that *E* is an enumerated type. Since *E* is a class, it is possible to create objects of type *TreeSet<E>* and *HashSet<E>*. However, because enums are so simple, trees and hash tables are not the most efficient implementation for sets of enumerated type values. Java provides the class `java.util.EnumSet` as an alternative way to create such sets.

Sets of enumerated type values are created using `static` methods in the class *EnumSet*. For example, if `e1`, `e2`, and `e3` are values belonging to the enumerated type *E*, then the method

```
EnumSet.of( e1, e2, e3 )
```

creates and returns a set of type *EnumSet<E>* that contains exactly the elements `e1`, `e2`, and `e3`. The set implements the interface *Set<E>*, so all the usual set and collection operations are available. The implementation of these operations is very efficient. The implementation uses what is called a *bit vector*. A bit is a quantity that has only two possible values, zero and one. A set of type *EnumSet<E>* is represented by a bit vector that contains one bit for each enum constant in the enumerated type *E*; the bit corresponding to the enum constant `e` is 1 if `e` is a member of the set and is 0 if `e` is not a member of the set. The bit vectors for two sets of type *EnumSet<E>* can be very easily combined to represent such operations as the union and intersection of two sets. The bit vector representation is feasible for *EnumSets*, but not for other sets in Java, because an enumerated type contains only a small finite number of enum constants. (Java actually has a class named *BitSet* that uses bit vectors to represent finite sets of non-negative integers, but this class is not part of the Java Collection Framework and does not implement the *Set* interface.)

The function `EnumSet.of` can be used with any positive number of parameters. All the parameters must be values of the same enumerated type. Null values are not allowed. An *EnumSet* cannot contain the value `null`—any attempt to add `null` to an *EnumSet* will result in a *NullPointerException*.

There is also a function `EnumSet.range(e1,e2)` that returns an *EnumSet* consisting of the enum constants between `e1` and `e2`, inclusive. The ordering of enum constants is the same as the order in which they are listed in the definition of the enum. In `EnumSet.range(e1,e2)`, `e1` and `e2` must belong to the same enumerated type, and `e1` must be less than or equal to `e2`.

If *E* is an enum, then `EnumSet.allOf(E.class)` is a set that contains all values of type *E*. `EnumSet.noneOf(E.class)` is an ***empty set***, a set of type *EnumSet<E>* that contains no elements at all. Note that in `EnumSet.allOf(E.class)` and `EnumSet.noneOf(E.class)`, the odd-looking parameter represents the enumerated type class itself. If `eset` is a set of type *EnumSet<E>*, then `EnumSet.complementOf(eset)` is a set that contains all the enum constants of *E* that are **not** in `eset`.

As an example, consider a program that keeps schedules of events. The program must keep track of repeating events that happen on specified days of the week. For example, an event might take place only on weekdays, or only on Wednesdays and Fridays. In other words, associated with the event is the **set** of days of the week on which it takes place. This information can be represented using the enumerated type

```
enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY }
```

The days of the week on which an event takes place would then be a value of type *EnumSet<Day>*. An object of type *RepeatingEvent* would have an instance variable of type *EnumSet<Day>* to hold this information. An event that takes place on Wednesdays and Fridays would have the associated set

```
EnumSet.of( Day.WEDNESDAY, Day.FRIDAY )
```

We could define some common sets of *Days* as

```
EnumSet<Day> weekday = EnumSet.range( Day.MONDAY, Day.FRIDAY );
EnumSet<Day> weekend = EnumSet.complementOf( weekday );
EnumSet<Day> everyday = EnumSet.allOf( Day.class );
```

EnumSets are often used to specify sets of "options" that are to be applied during some type of processing. For example, a program that draws characters in fancy fonts might have various options that can be applied. Let's say that the options are bold, italic, underlined, strikethrough, and boxed. Note that we are assuming that options can be combined in arbitrary ways. For example, you can have italic, boxed, underlined characters. This just means that we need to keep track of a **set** of options. If the options are represented by the enumerated type

```
enum FontOption { BOLD, ITALIC, UNDERLINED, STRIKETHROUGH, BOXED }
```

then a set of options is represented by a value of type *EnumSet<FontOption>*. Suppose that `options` is a variable of this type that represents the set of options that are currently being applied by the program. Then we can do things like:

- `options = EnumSet.noneOf( FontOption.class )` — Turn off all options.
- `options = EnumSet.of( FontOption.BOLD )` — Use bold, with no other options.
- `options.add( FontOption.BOLD )` — Add bold to any options that are already on.
- `options.remove( FontOption.UNDERLINED )` — Turn underlining off (if it's on).

This is a nice, safe way to work with sets of options. Applications like this are one of the major reasons that enumerated types were introduced.

10.3 Maps

An array of N elements can be thought of as a way of associating some item with each of the integers `0`, `1`, ..., `N-1`. If `i` is one of these integers, it's possible to ***get*** the item associated with `i`, and it's possible to ***put*** a new item in the `i`-th position. These "get" and "put" operations define what it means to be an array.

A ***map*** is a kind of generalized array. Like an array, a map is defined by "get" and "put" operations. But in a map, these operations are defined not for integers `0`, `1`, ..., `N-1`, but for arbitrary objects of some specified type *T*. Associated to these objects of type *T* are objects of some possibly different type *S*.

In fact, some programming languages use the term ***associative array*** instead of "map" and use the same notation for associative arrays as for regular arrays. In those languages, for example, you might see the notation `A["fred"]` used to indicate the item associated to the string "fred" in the associative array `A`. Java does not use array notation for maps, unfortunately, but the idea is the same: A map is like an array, but the indices for a map are objects, not integers. In a map, an object that serves as an "index" is called a ***key***. The object that is

associated with a key is called a ***value***. Note that a key can have at most one associated value, but the same value can be associated to several different keys. A map can be considered to be a set of "associations," where each association is a key/value pair.

10.3.1 The Map Interface

In Java, maps are defined by the interface `java.util.Map`, which includes `put` and `get` methods as well as other general methods for working with maps. The map interface, *Map<K,V>*, is parameterized by **two** types. The first type parameter, *K*, specifies the type of objects that are possible keys in the map; the second type parameter, *V*, specifies the type of objects that are possible values in the map. For example, a map of type *Map<Date,JButton>* would associate values of type *JButton* to keys of type *Date*. For a map of type *Map<String,String>*, both the keys and the values are of type *String*.

Suppose that `map` is a variable of type *Map<K,V>* for some specific types *K* and *V*. Then the following are some of the methods that are defined for `map`:

- `map.get(key)` — returns the object of type *V* that is associated by the map to the `key`. `key` can be any object; it does not have to be of type *K*. If the map does not associate any value with `key`, then the return value is `null`. Note that it's also possible for the return value to be `null` when the map explicitly associates the value `null` with the key. Referring to "`map.get(key)`" is similar to referring to "`A[key]`" for an array `A`. (But note that there is nothing like an *IndexOutOfBoundsException* for maps.)
- `map.put(key,value)` — Associates the specified `value` with the specified `key`, where `key` must be of type *K* and `value` must be of type *V*. If the map already associated some other value with the key, then the new value replaces the old one. This is similar to the command "`A[key] = value`" for an array.
- `map.putAll(map2)` — if `map2` is another map of type *Map<K,V>*, this copies all the associations from `map2` into map.
- `map.remove(key)` — if `map` associates a value to the specified `key`, that association is removed from the map. `key` can be any object; it does not have to be of type *K*.
- `map.containsKey(key)` — returns a boolean value that is `true` if the map associates some value to the specified `key`. `key` can be any object; it does not have to be of type *K*.
- `map.containsValue(value)` — returns a boolean value that is `true` if the map associates the specified `value` to some key. `value` can be any object; it does not have to be of type *V*.
- `map.size()` — returns an **int** that gives the number of key/value associations in the map.
- `map.isEmpty()` — returns a boolean value that is `true` if the map is empty, that is if it contains no associations.
- `map.clear()` — removes all associations from the map, leaving it empty.

The `put` and `get` methods are certainly the most commonly used of the methods in the *Map* interface. In many applications, these are the only methods that are needed, and in such cases a map is really no more difficult to use than a standard array.

Java includes two classes that implement the interface *Map<K,V>*: *TreeMap<K,V>* and *HashMap<K,V>*. In a *TreeMap*, the key/value associations are stored in a sorted tree, in which they are sorted according to their `keys`. For this to work, it must be possible to compare the keys to one another. This means either that the keys must implement the interface *Comparable<K>*, or that a *Comparator* must be provided for comparing keys. (The *Comparator* can be

provided as a parameter to the *TreeMap* constructor.) Note that in a *TreeMap*, as in a *TreeSet*, the `compareTo()` method is used to decide whether two keys are to be considered the same. This can have undesirable consequences if the `compareTo()` method does not agree with the usual notion of equality, and you should keep this in mind when using *TreeMaps*.

A *HashMap* does not store associations in any particular order, so the keys that can be used in a *HashMap* do not have to be comparable. However, the key class should have reasonable definitions for the `equals()` method and for a `hashCode()` method that is discussed later in this section; most of Java's standard classes define these methods correctly. Most operations are a little faster on *HashMaps* than they are on *TreeMaps*. In general, you should use a *HashMap* unless you have some particular need for the ordering property of a *TreeMap*. In particular, if you are only using the `put` and `get` operations, you can safely use a *HashMap*.

Let's consider an example where maps would be useful. In Subsection 7.4.2, I presented a simple *PhoneDirectory* class that associated phone numbers with names. That class defined operations `addEntry(name,number)` and `getNumber(name)`, where both `name` and `number` are given as *Strings*. In fact, the phone directory is acting just like a map, with the `addEntry` method playing the role of the `put` operation and `getNumber` playing the role of `get`. In a real programming application, there would be no need to define a new class; we could simply use a map of type *Map<String,String>*. A directory would be defined as

```
Map<String,String> directory = new Map<String,String>();
```

and then `directory.put(name,number)` would record a phone number in the directory and `directory.get(name)` would retrieve the phone number associated with a given name.

10.3.2 Views, SubSets, and SubMaps

A *Map* is not a *Collection*, and maps do not implement all the operations defined on collections. In particular, there are no iterators for maps. Sometimes, though, it's useful to be able to iterate through all the associations in a map. Java makes this possible in a roundabout but clever way. If `map` is a variable of type *Map<K,V>*, then the method

```
map.keySet()
```

returns the set of all objects that occur as keys for associations in the map. The value returned by this method is an object that implements the interface *Set<K>*. The elements of this set are the map's keys. The obvious way to implement the `keySet()` method would be to create a new set object, add all the keys from the map, and return that set. But that's not how it's done. The value returned by `map.keySet()` is not an independent object. It is what is called a ***view*** of the actual objects that are stored in the map. This "view" of the map implements the *Set<K>* interface, but it does it in such a way that the methods defined in the interface refer directly to keys in the map. For example, if you remove a key from the view, that key—along with its associated value—is actually removed from the map. It's not legal to add an object to the view, since it doesn't make sense to add a key to a map without specifying the value that should be associated to the key. Since `map.keySet()` does not create a new set, it's very efficient, even for very large maps.

One of the things that you can do with a *Set* is get an *Iterator* for it and use the iterator to visit each of the elements of the set in turn. We can use an iterator (or a for-each loop) for the key set of a map to traverse the map. For example, if `map` is of type *Map<String,Double>*, we could write:

```
Set<String> keys = map.keySet();     // The set of keys in the map.
Iterator<String> keyIter = keys.iterator();
System.out.println("The map contains the following associations:");
while (keyIter.hasNext()) {
   String key = keyIter.next();  // Get the next key.
   Double value = map.get(key);  // Get the value for that key.
   System.out.println( "   (" + key + "," + value + ")" );
}
```

Or we could do the same thing more easily, avoiding the explicit use of an iterator, with a for-each loop:

```
System.out.println("The map contains the following associations:");
for ( String key : map.keySet() ) {  // "for each key in the map's key set"
    Double value = map.get(key);
    System.out.println( "   (" + key + "," + value + ")" );
}
```

If the map is a *TreeMap*, then the key set of the map is a sorted set, and the iterator will visit the keys in ascending order. For a *HashMap*, the keys are visited in an arbitrary, unpredictable order.

The *Map* interface defines two other views. If `map` is a variable of type *Map<K,V>*, then the method:

```
map.values()
```

returns an object of type *Collection<V>* that contains all the values from the associations that are stored in the map. The return value is a *Collection* rather than a *Set* because it can contain duplicate elements (since a map can associate the same value to any number of keys). The method:

```
map.entrySet()
```

returns a set that contains all the associations from the map. The elements in the set are objects of type *Map.Entry<K,V>*. *Map.Entry<K,V>* is defined as a static nested interface inside the interface *Map<K,V>*, so its full name contains a period. However, the name can be used in the same way as any other type name. (The return type of the method `map.entrySet()` is written as *Set<Map.Entry<K,V>>*. The type parameter in this case is itself a parameterized type. Although this might look confusing, it's just Java's way of saying that the elements of the set are of type *Map.Entry<K,V>*.) The information in the set returned by `map.entrySet()` is actually no different from the information in the map itself, but the set provides a different view of this information, with different operations. Each `Map.Entry` object contains one key/value pair, and defines methods `getKey()` and `getValue()` for retrieving the key and the value. There is also a method, `setValue(value)`, for setting the value; calling this method for a `Map.Entry` object will modify the map itself, just as if the map's `put` method were called. As an example, we can use the entry set of a map to print all the key/value pairs in the map. This is more efficient than using the key set to print the same information, as I did in the above example, since we don't have to use the `get()` method to look up the value associated with each key. Suppose again that `map` is of type *Map<String,Double>*. Then we can write:

```
Set<Map.Entry<String,Double>> entries = map.entrySet();
Iterator<Map.Entry<String,Double>> entryIter = entries.iterator();
System.out.println("The map contains the following associations:");
while (entryIter.hasNext()) {
```

```
   Map.Entry<String,Double> entry = entryIter.next();
   String key = entry.getKey();  // Get the key from the entry.
   Double value = entry.getValue();  // Get the value.
   System.out.println( "   (" + key + "," + value + ")" );
}
```

or, using a for-each loop:

```
System.out.println("The map contains the following associations:");
for ( Map.Entry<String,Double> entry : map.entrySet() )
   System.out.println( "   (" + entry.getKey() + "," + entry.getValue() + ")" );
```

* * *

Maps are not the only place in Java's generic programming framework where views are used. For example, the interface *List<T>* defines a ***sublist*** as a view of a part of a list. If `list` implements the interface *List<T>*, then the method:

```
list.subList( fromIndex, toIndex )
```

where `fromIndex` and `toIndex` are integers, returns a view of the part of the list consisting of the list elements in positions between `fromIndex` and `toIndex` (including `fromIndex` but excluding `toIndex`). This view lets you operate on the sublist using any of the operations defined for lists, but the sublist is not an independent list. Changes made to the sublist are actually made to the original list.

Similarly, it is possible to obtain views that represent certain subsets of a sorted set. If `set` is of type *TreeSet<T>*, then `set.subSet(fromElement,toElement)` returns a *Set<T>* that contains all the elements of `set` that are between `fromElement` and `toElement` (including `fromElement` and excluding `toElement`). The parameters `fromElement` and `toElement` must be objects of type *T*. For example, if `words` is a set of type *TreeSet<String>* in which all the elements are strings of lower case letters, then `words.subSet("m","n")` contains all the elements of `words` that begin with the letter 'm'. This subset is a view of part of the original set. That is, creating the subset does not involve copying elements. And changes made to the subset, such as adding or removing elements, are actually made to the original set. The view `set.headSet(toElement)` consists of all elements from the set which are strictly less than `toElement`, and `set.tailSet(fromElement)` is a view that contains all elements from the set that are greater than or equal to `fromElement`.

The class *TreeMap<K,V>* defines three submap views. A submap is similar to a subset. A submap is a *Map* that contains a subset of the keys from the original *Map*, along with their associated values. If `map` is a variable of type *TreeMap<K,V>*, and if `fromKey` and `toKey` are of type *T*, then `map.subMap(fromKey,toKey)` returns a view that contains all key/value pairs from `map` whose keys are between `fromKey` and `toKey` (including `fromKey` and excluding `toKey`). There are also views `map.headMap(toKey)` and `map.tailMap(fromKey)` which are defined analogously to `headSet` and `tailSet`. Suppose, for example, that `blackBook` is a map of type *TreeMap<String,String>* in which the keys are names and the values are phone numbers. We can print out all the entries from `blackBook` where the name begins with "M" as follows:

```
Map<String,String> ems = blackBook.subMap("M","N");
     // This submap contains entries for which the key is greater
     // than or equal to "M" and strictly less than "N".

if (ems.isEmpty()) {
   System.out.println("No entries beginning with M.");
}
```

```
else {
   System.out.println("Entries beginning with M:");
   for ( Map.Entry<String,String> entry : ems.entrySet() )
      System.out.println( "   " + entry.getKey() + ": " + entry.getValue() );
}
```

Subsets and submaps are probably best thought of as generalized search operations that make it possible to find all the items in a range of values, rather than just to find a single value. Suppose, for example that a database of scheduled events is stored in a map of type *TreeMap<Date,Event>* in which the keys are the times of the events, and suppose you want a listing of all events that are scheduled for some time on July 4, 2011. Just make a submap containing all keys in the range from 12:00 AM, July 4, 2011 to 12:00 AM, July 5, 2011, and output all the entries from that submap. This type of search, which is known as a ***subrange query*** is quite common.

10.3.3 Hash Tables and Hash Codes

HashSets and *HashMaps* are implemented using a data structure known as a ***hash table***. You don't need to understand hash tables to use *HashSets* or *HashMaps*, but any computer programmer should be familiar with hash tables and how they work.

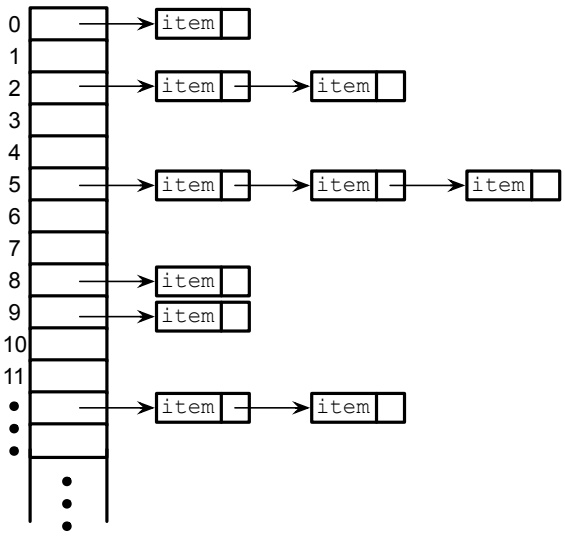
Hash tables are an elegant solution to the search problem. A hash table, like a *HashMap*, stores key/value pairs. Given a key, you have to search the table for the corresponding key/value pair. When a hash table is used to implement a set, the values are all null, and the only question is whether or not the key occurs in the set. You still have to search for the key to check whether it is there or not.

In most search algorithms, in order to find the item you are interested in, you have to look through a bunch of other items that don't interest you. To find something in an unsorted list, you have to go though the items one-by-one until you come to the one you are looking for. In a binary sort tree, you have to start at the root and move down the tree until you find the item you want. When you search for a key/value pair in a hash table, you can go directly to the location that contains the item you want. You don't have to look through any other items. (This is not quite true, but it's close.) The location of the key/value pair is computed from the key: You just look at the key, and then you go directly to the location where it is stored.

How can this work? If the keys were integers in the range 0 to 99, we could store the key/value pairs in an array, `A`, of 100 elements. The key/value pair with key `K` would be stored in `A[K]`. The key takes us directly to the location of the key/value pair. The problem is that there are usually far too many different possible keys for us to be able to use an array with one location for each possible key. For example, if the key can be any value of type **int**, then we would need an array with over four billion locations—quite a waste of space if we are only going to store, say, a few thousand items! If the key can be a string of any length, then the number of possible keys is infinite, and using an array with one location for each possible key is simply impossible.

Nevertheless, hash tables store their data in an array, and the array index where a key is stored is based on the key. The index is not equal to the key, but it is computed from the key. The array index for a key is called the ***hash code*** for that key. A function that computes a hash code, given a key, is called a ***hash function***. To find a key in a hash table, you just have to compute the hash code of the key and go directly to the array location given by that hash code. If the hash code is 17, look in array location number 17.

Now, since there are fewer array locations than there are possible keys, it's possible that we might try to store two or more keys in the same array location. This is called a ***collision***. A collision is not an error. We can't reject a key just because another key happened to have the same hash code. A hash table must be able to handle collisions in some reasonable way. In the type of hash table that is used in Java, each array location actually holds a linked list of key/value pairs (possibly an empty list). When two items have the same hash code, they are in the same linked list. The structure of the hash table looks something like this:



In this diagram, there is one item with hash code 0, no items with hash code 1, two items with hash code 2, and so on. In a properly designed hash table, most of the linked lists are of length zero or one, and the average length of the lists is less than one. Although the hash code of a key doesn't necessarily take you directly to that key, there are probably no more than one or two other items that you have to look through before finding the key you want. For this to work properly, the number of items in the hash table should be somewhat less than the number of locations in the array. In Java's implementation, whenever the number of items exceeds 75% of the array size, the array is replaced by a larger one and all the items in the old array are inserted into the new one. (This is why adding one new item will sometimes cause the ordering of all the items in the hash table to change completely.)

There is still the question of where hash codes come from. Every object in Java has a hash code. The *Object* class defines the method `hashCode()`, which returns a value of type **int**. When an object, `obj`, is stored in a hash table that has `N` locations, a hash code in the range `0` to `N-1` is needed. This hash code is computed as `Math.abs(obj.hashCode()) % N`, the remainder when the absolute value of `obj.hashCode()` is divided by `N`. (The `Math.abs` is necessary because `obj.hashCode()` can be a negative integer, and we need a non-negative number to use as an array index.)

For hashing to work properly, two objects that are equal according to the `equals()` method must have the same hash code. In the *Object* class, this condition is satisfied because both `equals()` and `hashCode()` are based on the address of the memory location where the object is stored. However, as noted in Subsection 10.1.6, many classes redefine the `equals()` method. If a class redefines the `equals()` method, and if objects of that class will be used as keys in hash tables, then the class should also redefine the `hashCode()` method. For example, in the

String class, the `equals()` method is redefined so that two objects of type *String* are considered to be equal if they contain the same sequence of characters. The `hashCode()` method is also redefined in the *String* class, so that the hash code of a string is computed from the characters in that string rather than from its location in memory. For Java's standard classes, you can expect `equals()` and `hashCode()` to be correctly defined. However, you might need to define these methods in classes that you write yourself.

Writing a good hash function is something of an art. In order to work well, the hash function must spread the possible keys fairly evenly over the hash table. Otherwise, the items in a table can be concentrated in a subset of the available locations, and the linked lists at those locations can grow to large size; that would destroy the efficiency that is the major reason for hash tables to exist in the first place. However, I won't cover techniques for creating good hash functions in this book.

10.4 Programming with the Java Collection Framework

In this section, we'll look at some programming examples that use classes from the Java Collection Framework. The Collection Framework is easy to use, especially compared to the difficulty of programming new data structures from scratch.

10.4.1 Symbol Tables

We begin with a straightforward but important application of maps. When a compiler reads the source code of a program, it encounters definitions of variables, subroutines, and classes. The names of these things can be used later in the program. The compiler has to remember the definition of each name, so that it can recognize the name and apply the definition when the name is encountered later in the program. This is a natural application for a *Map*. The name can be used as a key in the map. The value associated to the key is the definition of the name, encoded somehow as an object. A map that is used in this way is called a ***symbol table***.

In a compiler, the values in a symbol table can be quite complicated, since the compiler has to deal with names for various sorts of things, and it needs a different type of information for each different type of name. We will keep things simple by looking at a symbol table in another context. Suppose that we want a program that can evaluate expressions entered by the user, and suppose that the expressions can contain variables, in addition to operators, numbers, and parentheses. For this to make sense, we need some way of assigning values to variables. When a variable is used in an expression, we need to retrieve the variable's value. A symbol table can be used to store the data that we need. The keys for the symbol table are variable names. The value associated with a key is the value of that variable, which is of type **double**. The symbol table will be an object of type *Map<String,Double>*. (Remember that primitive types such as **double** can't be used as type parameters; a wrapper class such as *Double* must be used instead. See Subsection 10.1.7.)

To demonstrate the idea, we'll use a rather simple-minded program in which the user types commands such as:

```
let x = 3 + 12
print 2 + 2
print 10*x +17
let rate = 0.06
print 1000*(1+rate)
```

The program is an interpreter for a very simple language. The only two commands that the program understands are "print" and "let". When a "print" command is executed, the computer evaluates the expression and displays the value. If the expression contains a variable, the computer has to look up the value of that variable in the symbol table. A "let" command is used to give a value to a variable. The computer has to store the value of the variable in the symbol table. (Note: The "variables" I am talking about here are not variables in the Java program. The Java program is executing a sort of program typed in by the user. I am talking about variables in the user's program. The user gets to make up variable names, so there is no way for the Java program to know in advance what the variables will be.)

In Subsection 9.5.2, we saw how to write a program, *SimpleParser2.java*, that can evaluate expressions that do not contain variables. Here, I will discuss another example program, *SimpleInterpreter.java*, that is based on the older program. I will only talk about the parts that are relevant to the symbol table.

The program uses a *HashMap* as the symbol table. A *TreeMap* could also be used, but since the program does not need to access the variables in alphabetical order, we don't need to have the keys stored in sorted order. The symbol table in the program is represented by a variable named `symbolTable` of type *HashMap<String,Double>*. At the beginning of the program, the symbol table object is created with the command:

```
symbolTable = new HashMap<String,Double>();
```

This creates a map that initially contains no key/value associations. To execute a "let" command, the program uses the symbol table's `put()` method to associate a value with the variable name. Suppose that the name of the variable is given by a *String*, `varName`, and the value of the variable is stored in a variable `val` of type **double**. The following command would then set the value associated with the variable in the symbol table:

```
symbolTable.put( varName, val );
```

In the program *SimpleInterpreter.java*, you'll find this in the method named `doLetCommand()`. The actual value that is stored in the symbol table is an object of type *Double*. We can use the **double** value `val` in the call to `put` because Java does an automatic conversion of type **double** to *Double* when necessary. The **double** value is "wrapped" in an object of type *Double*, so that, in effect, the above statement is equivalent to

```
symbolTable.put( varName, new Double(val) );
```

Just for fun, I decided to pre-define two variables named "pi" and "e" whose values are the usual mathematical constants π and e. In Java, the values of these constants are given by `Math.PI` and `Math.E`. To make these variables available to the user of the program, they are added to the symbol table with the commands:

```
symbolTable.put( "pi", Math.PI );
symbolTable.put( "e", Math.E );
```

When the program encounters a variable while evaluating an expression, the symbol table's `get()` method is used to retrieve its value. The function `symbolTable.get(varName)` returns a value of type *Double*. It is possible that the return value is `null`; this will happen if no value has ever been assigned to `varName` in the symbol table. It's important to check this possibility. It indicates that the user is trying to use a variable that the user has not defined. The program considers this to be an error, so the processing looks something like this:

```
Double val = symbolTable.get(varName);
if (val == null) {
   ... // Throw an exception:  Undefined variable.
}
// The value associated to varName is val.doubleValue()
```

You will find this code, more or less, in a method named `primaryValue()` in *SimpleInterpreter.java*.

As you can see from this example, *Maps* are very useful and are really quite easy to use.

10.4.2 Sets Inside a Map

The objects in a collection or map can be of any type. They can even be collections. Here's an example where it's natural to store sets as the value objects in a map.

Consider the problem of making an index for a book. An index consists of a list of terms that appear in the book. Next to each term is a list of the pages on which that term appears. To represent an index in a program, we need a data structure that can hold a list of terms, along with a list of pages for each term. Adding new data should be easy and efficient. When it's time to print the index, it should be easy to access the terms in alphabetical order. There are many ways this could be done, but I'd like to use Java's generic data structures and let them do as much of the work as possible.

We can think of an index as a *Map* that associates a list of page references to each term. The terms are keys, and the value associated with a given key is the list of page references for that term. A *Map* can be either a *TreeMap* or a *HashMap*, but only a *TreeMap* will make it easy to access the terms in sorted order. The value associated with a term is a list of page references. How can we represent such a value? If you think about it, you see that it's not really a list in the sense of Java's generic classes. If you look in any index, you'll see that a list of page references has no duplicates, so it's really a set rather than a list. Furthermore, the page references for a given term are always printed in increasing order, so we want a sorted set. This means that we should use a *TreeSet* to represent each list of page references. The values that we really want to put in this set are of type **int**, but once again we have to deal with the fact that generic data structures can only hold objects, so we must use the wrapper class, *Integer*, for the objects in the set.

To summarize, an index will be represented by a *TreeMap*. The keys for the map will be terms, which are of type *String*. The values in the map will be *TreeSets* that contain *Integers* that are the page numbers of every page on which a term appears. The parameterized type that we should use for the sets is *TreeSet<Integer>*. For the *TreeMap* that represents the index as a whole, the key type is *String* and the value type is *TreeSet<Integer>*. This means that the index has type

```
TreeMap< String, TreeSet<Integer> >
```

This is just the usual *TreeMap<K,V>* with `K=String` and `V=TreeSet<Integer>`. A type name as complicated as this one can look intimidating (especially, I think, when used in a constructor with the `new` operator), but if you think about the data structure that we want to represent, it makes sense. Given a little time and practice, you can get used to types like this one.

To make an index, we need to start with an empty *TreeMap* and look through the book, inserting every reference that we want to be in the index into the map. We then need to print out the data from the map. Let's leave aside the question of how we find the references to put in the index, and just look at how the *TreeMap* is used. It can be created with the commands:

```
TreeMap<String,TreeSet<Integer>>  index;         // Declare the variable.
index = new TreeMap<String,TreeSet<Integer>>();  // Create the map object.
```

Now, suppose that we find a reference to some `term` (of type *String*) on some `pageNum` (of type **int**). We need to insert this information into the index. To do this, we should look up the term in the index, using `index.get(term)`. The return value is either `null` or is the set of page references that we have previously found for the term. If the return value is `null`, then this is the first page reference for the term, so we should add the term to the index, with a new set that contains the page reference we've just found. If the return value is non-`null`, we already have a set of page references, and we should just add the new page reference to the set. Here is a subroutine that does this:

```
/**
 * Add a page reference to the index.
 */
void addReference(String term, int pageNum) {
   TreeSet<Integer> references; // The set of page references that we
                                //    have so far for the term.
   references = index.get(term);
   if (references == null){
          // This is the first reference that we have
          // found for the term.  Make a new set containing
          // the page number and add it to the index, with
          // the term as the key.
       TreeSet<Integer> firstRef = new TreeSet<Integer>();
       firstRef.add( pageNum );  // pageNum is "autoboxed" to give an Integer!
       index.put(term,firstRef);
   }
   else {
         // references is the set of page references
         // that we have found previously for the term.
         // Add the new page number to that set.  This
         // set is already associated to term in the index.
      references.add( pageNum ); // pageNum is "autoboxed" to give an Integer!
   }
}
```

The only other thing we need to do with the index is print it out. We want to iterate through the index and print out each term, together with the set of page references for that term. We could use an *Iterator* to iterate through the index, but it's much easier to do it with a for-each loop. The loop will iterate through the entry set of the map (see Subsection 10.3.2). Each "entry" is a key/value pair from the map; the key is a term and the value is the associated set of page references. Inside the for-each loop, we will have to print out a set of *Integers*, which can also be done with a for-each loop. So, here we have an example of nested for-each loops. (You might try to do the same thing entirely with iterators; doing so should give you some appreciation for the for-each loop!) Here is a subroutine that will print the index:

```
/**
 * Print each entry in the index.
 */
void printIndex() {

    for ( Map.Entry<String,TreeSet<Integer>>  entry :  index.entrySet() ) {
```

```
            String term = entry.getKey();
            TreeSet<Integer> pageSet = entry.getValue();

            System.out.print( term + " " );
            for ( int page : pageSet ) {
                System.out.print( page + " " );
            }
            System.out.println();

        }

    }
```

The hardest thing here is the name of the type `Map.Entry<String,TreeSet<Integer>>`! Remember that the entries in a map of type *Map<K,V>* have type *Map.Entry<K,V>*, so the type parameters in `Map.Entry<String,TreeSet<Integer>>` are simply copied from the declaration of `index`. Another thing to note is that I used a loop control variable, `page`, of type *int* to iterate through the elements of `pageSet`, which is of type *TreeSet<Integer>*. You might have expected `page` to be of type *Integer*, not **int**, and in fact *Integer* would have worked just as well here. However, **int** does work, because of automatic type conversion: it's legal to assign a value of type *Integer* to a variable of type **int**. (To be honest, I was sort of surprised that this worked when I first tried it!)

This is not a lot of code, considering the complexity of the operations. I have not written a complete indexing program, but Exercise 10.5 presents a problem that is almost identical to the indexing problem.

* * *

By the way, in this example, I would prefer to print each list of page references with the integers separated by commas. In the `printIndex()` method given above, they are separated by spaces. There is an extra space after the last page reference in the list, but it does no harm since it's invisible in the printout. An extra comma at the end of the list would be annoying. The lists should be in a form such as "`17,42,105`" and not "`17,42,105,`". The problem is, how to leave that last comma out. Unfortunately, this is not so easy to do with a for-each loop. It might be fun to look at a few ways to solve this problem. One alternative is to use an iterator:

```
Iterator<Integer>  iter = pageSet.iterator();
int firstPage = iter.next();  // In this program, we know the set has at least
                              // one element.  Note also that this statement
                              // uses an auto-conversion from Integer to int.
System.out.print(firstPage);
while ( iter.hasNext() ) {
   int nextPage = iter.next();
   System.out.print("," + nextPage);
}
```

Another possibility is to use the fact that the *TreeSet* class defines a method `first()` that returns the first item in the set, that is, the one that is smallest in terms of the ordering that is used to compare items in the set. (It also defines the method `last()`.) We can solve our problem using this method and a for-each loop:

```
int firstPage = pageSet.first();  // Find out the first page number in the set.
for ( int page : pageSet ) {
   if ( page != firstPage )
      System.out.print(","); // Output comma only if this is not the first page.
```

```
      System.out.print(page);
   }
```

Finally, here is an elegant solution using a subset view of the tree. (See Subsection 10.3.2.) Actually, this solution might be a bit extreme:

```
int firstPage = pageSet.first();  // Get first item, which we know exists.
System.out.print(firstPage);      // Print first item, with no comma.
for ( int page : pageSet.tailSet( firstPage+1 ) ) // Process remaining items.
   System.out.print( "," + page );
```

10.4.3 Using a Comparator

There is a potential problem with our solution to the indexing problem. If the terms in the index can contain both upper case and lower case letters, then the terms will **not** be in alphabetical order! The ordering on *String* is not alphabetical. It is based on the Unicode codes of the characters in the string. The codes for all the upper case letters are less than the codes for the lower case letters. So, for example, terms beginning with "Z" come before terms beginning with "a". If the terms are restricted to use lower case letters only (or upper case only), then the ordering would be alphabetical. But suppose that we allow both upper and lower case, and that we insist on alphabetical order. In that case, our index can't use the usual ordering for `Strings`. Fortunately, it's possible to specify a different method to be used for comparing the keys of a map. This is a typical use for a *Comparator*.

Recall that an object that implements the interface *Comparator<T>* defines a method for comparing two objects of type *T*:

```
public int compare( T obj1, T obj2 )
```

This method should return an integer that is positive, zero, or negative, depending on whether `obj1` is less than, equal to, or greater than `obj2`. We need an object of type *Comparator<String>* that will compare two *Strings* based on alphabetical order. The easiest way to do this is to convert the *Strings* to lower case and use the default comparison on the lower case *Strings*. The following class defines such a comparator:

```
/**
 * Represents a Comparator that can be used for comparing two
 * strings based on alphabetical order.
 */
class AlphabeticalOrder implements Comparator<String> {
   public int compare(String str1, String str2) {
      String s1 = str1.toLowerCase();  // Convert to lower case.
      String s2 = str2.toLowerCase();
      return s1.compareTo(s2);  // Compare lower-case Strings.
   }
}
```

To solve our indexing problem, we just need to tell our index to use an object of type *AlphabeticalOrder* for comparing keys. This is done by providing a *Comparator* object as a parameter to the constructor. We just have to create the `index` in our example with the command:

```
index = new TreeMap<String,TreeSet<Integer>>( new AlphabeticalOrder() );
```

This does work. However, I've been concealing one technicality. Suppose, for example, that the indexing program calls `addReference("aardvark",56)` and that it later calls `addReference("Aardvark",102)`. The words "aardvark" and "Aardvark" differ only in that one of them begins with an upper case letter; when converted to lower case, they are the same. When we insert them into the index, do they count as two different terms or as one term? The answer depends on the way that a *TreeMap* tests objects for equality. In fact, *TreeMaps* and *TreeSets* always use a *Comparator* object or a `compareTo` method to test for equality. They do **not** use the `equals()` method for this purpose. The *Comparator* that is used for the *TreeMap* in this example returns the value zero when it is used to compare "aardvark" and "Aardvark", so the *TreeMap* considers them to be the same. Page references to "aardvark" and "Aardvark" are combined into a single list, and when the index is printed it will contain only the first version of the word that was encountered by the program. This is probably acceptable behavior in this example. If not, some other technique must be used to sort the terms into alphabetical order.

10.4.4 Word Counting

The final example in this section also deals with storing information about words. The problem here is to make a list of all the words that occur in a file, along with the number of times that each word occurs. The file will be selected by the user. The output of the program will consist of two lists. Each list contains all the words from the file, along with the number of times that the word occurred. One list is sorted alphabetically, and the other is sorted according to the number of occurrences, with the most common words at the top and the least common at the bottom. The problem here is a generalization of Exercise 7.6, which asked you to make an alphabetical list of all the words in a file, without counting the number of occurrences.

My word counting program can be found in the file *WordCount.java*. As the program reads an input file, it must keep track of how many times it encounters each word. We could simply throw all the words, with duplicates, into a list and count them later. But that would require a lot of extra storage space and would not be very efficient. A better method is to keep a counter for each word. The first time the word is encountered, the counter is initialized to 1. On subsequent encounters, the counter is incremented. To keep track of the data for one word, the program uses a simple class that holds a word and the counter for that word. The class is a `static` nested class:

```
/**
 * Represents the data we need about a word:  the word and
 * the number of times it has been encountered.
 */
private static class WordData {
   String word;
   int count;
   WordData(String w) {
         // Constructor for creating a WordData object when
         // we encounter a new word.
      word = w;
      count = 1;  // The initial value of count is 1.
   }
} // end class WordData
```

The program has to store all the *WordData* objects in some sort of data structure. We want to be able to add new words efficiently. Given a word, we need to check whether a *WordData*

object already exists for that word, and if it does, we need to find that object so that we can increment its counter. A *Map* can be used to implement these operations. Given a word, we want to look up a *WordData* object in the *Map*. This means that the word is the **key**, and the *WordData* object is the **value.** (It might seem strange that the key is also one of the instance variables in the value object, but in fact this is probably the most common situation: The value object contains all the information about some entity, and the key is one of those pieces of information; the partial information in the key is used to retrieve the full information in the value object.) After reading the file, we want to output the words in alphabetical order, so we should use a *TreeMap* rather than a *HashMap*. This program converts all words to lower case so that the default ordering on `Strings` will put the words in alphabetical order. The data is stored in a variable named `words` of type *TreeMap<String,WordData>*. The variable is declared and the map object is created with the statement:

```
TreeMap<String,WordData> words = new TreeMap<String,WordData>();
```

When the program reads a word from a file, it calls `words.get(word)` to find out if that word is already in the map. If the return value is `null`, then this is the first time the word has been encountered, so a new *WordData* object is created and inserted into the map with the command `words.put(word, new WordData(word))`. If `words.get(word)` is not `null`, then its value is the *WordData* object for this word, and the program only has to increment the counter in that object. The program uses a method `readNextWord()`, which was given in Exercise 7.6, to read one word from the file. This method returns `null` when the end of the file is encountered. Here is the complete code segment that reads the file and collects the data:

```
String word = readNextWord();
while (word != null) {
   word = word.toLowerCase();  // convert word to lower case
   WordData data = words.get(word);
   if (data == null)
      words.put( word, new WordData(word) );
   else
      data.count++;
   word = readNextWord();
}
```

After reading the words and printing them out in alphabetical order, the program has to sort the words by frequency and print them again. To do the sorting using a generic algorithm, I defined a simple *Comparator* class for comparing two word objects according to their frequency counts. The class implements the interface *Comparator<WordData>*, since it will be used to compare two objects of type *WordData*:

```
/**
 * A comparator class for comparing objects of type WordData according to
 * their counts.  This is used for sorting the list of words by frequency.
 */
private static class CountCompare implements Comparator<WordData> {
   public int compare(WordData data1, WordData data2) {
      return data2.count - data1.count;
          // The return value is positive if data1.count < data2.count.
          // I.E., data1 comes after data2 in the ordering if there
          // were FEWER occurrences of data1.word than of data2.word.
          // The words are sorted according to decreasing counts.
   }
} // end class CountCompare
```

Given this class, we can sort the *WordData* objects according to frequency by first copying them into a list and then using the generic method `Collections.sort(list,comparator)`. The *WordData* objects that we need are the values in the map, `words`. Recall that `words.values()` returns a *Collection* that contains all the values from the map. The constructor for the *ArrayList* class lets you specify a collection to be copied into the list when it is created. So, we can use the following commands to create a list of type *ArrayList<WordData>* containing the word data and then sort that list according to frequency:

```
ArrayList<WordData> wordsByFrequency = new ArrayList<WordData>( words.values() );
Collections.sort( wordsByFrequency, new CountCompare() );
```

You should notice that these two lines replace a lot of code! It requires some practice to think in terms of generic data structures and algorithms, but the payoff is significant in terms of saved time and effort.

The only remaining problem is to print the data. We have to print the data from all the *WordData* objects twice, first in alphabetical order and then sorted according to frequency count. The data is in alphabetical order in the *TreeMap*, or more precisely, in the values of the *TreeMap*. We can use a for-each loop to print the data in the collection `words.values()`, and the words will appear in alphabetical order. Another for-each loop can be used to print the data in the list `wordsByFrequency`, and the words will be printed in order of decreasing frequency. Here is the code that does it:

```
TextIO.putln("List of words in alphabetical order"
      + " (with counts in parentheses):\n");
for ( WordData data : words.values() )
   TextIO.putln("   " + data.word + " (" + data.count + ")");

TextIO.putln("\n\nList of words by frequency of occurrence:\n");
for ( WordData data : wordsByFrequency )
   TextIO.putln("   " + data.word + " (" + data.count + ")");
```

You can find the complete word-counting program in the file *WordCount.java*. Note that for reading and writing files, it uses the file I/O capabilities of *TextIO.java*, which were discussed in Subsection 2.4.5.

By the way, if you run the WordCount program on a reasonably large file and take a look at the output, it will illustrate something about the `Collections.sort()` method. The second list of words in the output is ordered by frequency, but if you look at a group of words that all have the same frequency, you will see that the words in that group are in alphabetical order. The method `Collections.sort()` was applied to sort the words by frequency, but before it was applied, the words were already in alphabetical order. When `Collections.sort()` rearranged the words, it did not change the ordering of words that have the same frequency, so they were still in alphabetical order within the group of words with that frequency. This is because the algorithm used by `Collections.sort()` is a ***stable*** sorting algorithm. A sorting algorithm is said to be stable if it satisfies the following condition: When the algorithm is used to sort a list according to some property of the items in the list, then the sort does not change the relative order of items that have the same value of that property. That is, if item B comes after item A in the list before the sort, and if both items have the same value for the property that is being used as the basis for sorting, then item B will still come after item A after the sorting has been done. Neither SelectionSort nor QuickSort are stable sorting algorithms. Insertion sort is stable, but is not very fast. Merge sort, the sorting algorithm used by `Collections.sort()`, is both stable and fast.

I hope that the programming examples in this section have convinced you of the usefulness of the Java Collection Framework!

10.5 Writing Generic Classes and Methods

So far in this chapter, you have learned about using the generic classes and methods that are part of the Java Collection Framework. Now, it's time to learn how to write new generic classes and methods from scratch. Generic programming produces highly general and reusable code—it's very useful for people who write reusable software libraries to know how to do generic programming, since it enables them to write code that can be used in many different situations. Not every programmer needs to write reusable software libraries, but every programmer should know at least a little about how to do it. In fact, just to read the JavaDoc documentation for Java's standard generic classes, you need to know some of the syntax that is introduced in this section.

I will not cover every detail of generic programming in Java in this section, but the material presented here should be sufficient to cover the most common cases.

10.5.1 Simple Generic Classes

Let's start with an example that illustrates the motivation for generic programming. In Subsection 10.2.1, I remarked that it would be easy to use a *LinkedList* to implement a queue. (Queues were introduced in Subsection 9.3.2.) To ensure that the only operations that are performed on the list are the queue operations `enqueue`, `dequeue`, and `isEmpty`, we can create a new class that contains the linked list as a private instance variable. To implement queues of strings, for example, we can define the class:

```
class QueueOfStrings {
   private LinkedList<String> items = new LinkedList<String>();
   public void enqueue(String item) {
      items.addLast(item);
   }
   public String dequeue() {
      return items.removeFirst();
   }
   public boolean isEmpty() {
      return (items.size() == 0);
   }
}
```

This is a fine and useful class. But, if this is how we write queue classes, and if we want queues of *Integers* or *Doubles* or *JButtons* or any other type, then we will have to write a different class for each type. The code for all of these classes will be almost identical, which seems like a lot of redundant programming. To avoid the redundancy, we can write a **generic** *Queue* class that can be used to define queues of any type of object.

The syntax for writing the generic class is straightforward: We replace the specific type *String* with a type parameter such as *T*, and we add the type parameter to the name of the class:

```
class Queue<T> {
   private LinkedList<T> items = new LinkedList<T>();
   public void enqueue(T item) {
```

```
        items.addLast(item);
    }
    public T dequeue() {
        return items.removeFirst();
    }
    public boolean isEmpty() {
        return (items.size() == 0);
    }
}
```

Note that within the class, the type parameter *T* is used just like any regular type name. It's used to declare the return type for `dequeue`, as the type of the formal parameter `item` in `enqueue`, and even as the actual type parameter in *LinkedList<T>*. Given this class definition, we can use parameterized types such as *Queue<String>* and *Queue<Integer>* and *Queue<JButton>*. That is, the *Queue* class is used in exactly the same way as built-in generic classes like *LinkedList* and *HashSet*.

Note that you don't have to use "T" as the name of the type parameter in the definition of the generic class. Type parameters are like formal parameters in subroutines. You can make up any name you like in the **definition** of the class. The name in the definition will be replaced by an actual type name when the class is used to declare variables or create objects. If you prefer to use a more meaningful name for the type parameter, you might define the *Queue* class as:

```
class Queue<ItemType> {
    private LinkedList<ItemType> items = new LinkedList<ItemType>();
    public void enqueue(ItemType item) {
        items.addLast(item);
    }
    public ItemType dequeue() {
        return items.removeFirst();
    }
    public boolean isEmpty() {
        return (items.size() == 0);
    }
}
```

Changing the name from "`T`" to "`ItemType`" has absolutely no effect on the meaning of the class definition or on the way that *Queue* is used.

Generic interfaces can be defined in a similar way. It's also easy to define generic classes and interfaces that have two or more type parameters, as is done with the standard interface *Map<T,S>*. A typical example is the definition of a "Pair" that contains two objects, possibly of different types. A simple version of such a class can be defined as:

```
class Pair<T,S> {
    public T first;
    public S second;
    public Pair( T a, S b ) {  // Constructor.
        first = a;
        second = b;
    }
}
```

This class can be used to declare variables and create objects such as:

```
Pair<String,Color> colorName = new Pair<String,Color>("Red", Color.RED);
Pair<Double,Double> coordinates = new Pair<Double,Double>(17.3,42.8);
```

Note that in the definition of the constructor in this class, the name "`Pair`" does **not** have type parameters. You might have expected "`Pair<T,S>`". However, the name of the class is "`Pair`", not "`Pair<T,S>`", and within the definition of the class, "`T`" and "`S`" are used as if they are the names of specific, actual types. Note in any case that type parameters are **never** added to the names of methods or constructors, only to the names of classes and interfaces.

10.5.2 Simple Generic Methods

In addition to generic classes, Java also has generic methods. An example is the method `Collections.sort()`, which can sort collections of objects of any type. To see how to write generic methods, let's start with a non-generic method for counting the number of times that a given string occurs in an array of strings:

```
/**
 * Returns the number of times that itemToCount occurs in list.  Items in the
 * list are tested for equality using itemToCount.equals(), except in the
 * special case where itemToCount is null.
 */
public static int countOccurrences(String[] list, String itemToCount) {
   int count = 0;
   if (itemToCount == null) {
      for ( String listItem : list )
         if (listItem == null)
            count++;
   }
   else {
      for ( String listItem : list )
         if (itemToCount.equals(listItem))
            count++;
   }
   return count;
}
```

Once again, we have some code that works for type *String*, and we can imagine writing almost identical code to work with other types of objects. By writing a generic method, we get to write a single method definition that will work for objects of any type. We need to replace the specific type *String* in the definition of the method with the name of a type parameter, such as *T*. However, if that's the only change we make, the compiler will think that "T" is the name of an actual type, and it will mark it as an undeclared identifier. We need some way of telling the compiler that "T" is a type parameter. That's what the "`<T>`" does in the definition of the generic class "`class Queue<T> { ...`". For a generic method, the "`<T>`" goes just before the name of the return type of the method:

```
public static <T> int countOccurrences(T[] list, T itemToCount) {
   int count = 0;
   if (itemToCount == null) {
      for ( T listItem : list )
         if (listItem == null)
            count++;
   }
   else {
      for ( T listItem : list )
         if (itemToCount.equals(listItem))
```

```
            count++;
      }
      return count;
   }
```

The "<T>" marks the method as being generic and specifies the name of the type parameter that will be used in the definition. Of course, the name of the type parameter doesn't have to be "T"; it can be anything. (The "<T>" looks a little strange in that position, I know, but it had to go somewhere and that's just where the designers of Java decided to put it.)

Given the generic method definition, we can apply it to objects of any type. If `wordList` is a variable of type String[] and `word` is a variable of type *String*, then

```
int ct = countOccurrences( wordList, word );
```

will count the number of times that `word` occurs in `wordList`. If `palette` is a variable of type Color[] and `color` is a variable of type *Color*, then

```
int ct = countOccurrences( palette, color );
```

will count the number of times that `color` occurs in `palette`. If `numbers` is a variable of type Integer[], then

```
int ct = countOccurrences( numbers, 17 );
```

will count the number of times that `17` occurs in `numbers`. This last example uses autoboxing; the 17 is automatically converted to a value of type *Integer*, as if we had said "`countOccurrences( numbers, new Integer(17) )`". Note that, since generic programming in Java applies only to objects, we **cannot** use `countOccurrences` to count the number of occurrences of 17 in an array of type int[].

A generic method can have one or more type parameters, such as the "T" in `countOccurrences`. Note that when a generic method is used, as in the function call "`countOccurrences(wordlist, word)`", there is no explicit mention of the type that is substituted for the type parameter. The compiler deduces the type from the types of the actual parameters in the method call. Since `wordlist` is of type String[], the compiler can tell that in "`countOccurrences(wordlist, word)`", the type that replaces *T* is *String*. This contrasts with the use of generic classes, as in "`new Queue<String>()`", where the type parameter is specified explicitly.

The `countOccurrences` method operates on an array. We could also write a similar method to count occurrences of an object in any collection:

```
public static <T> int countOccurrences(Collection<T> collection, T itemToCount) {
   int count = 0;
   if (itemToCount == null) {
      for ( T item : collection )
         if (item == null)
            count++;
   }
   else {
      for ( T item : collection )
         if (itemToCount.equals(item))
            count++;
   }
   return count;
}
```

Since *Collection<T>* is itself a generic type, this method is very general. It can operate on an *ArrayList* of *Integers*, a *TreeSet* of *Strings*, a *LinkedList* of *JButtons*, ...

10.5.3 Type Wildcards

There is a limitation on the sort of generic classes and methods that we have looked at so far: The type parameter in our examples, usually named *T*, can be any type at all. This is OK in many cases, but it means that the only things that you can do with *T* are things that can be done with **every** type, and the only things that you can do with objects of type *T* are things that you can do with **every** object. With the techniques that we have covered so far, you can't, for example, write a generic method that compares objects with the `compareTo()` method, since that method is not defined for all objects. The `compareTo()` method is defined in the *Comparable* interface. What we need is a way of specifying that a generic class or method only applies to objects of type *Comparable* and not to arbitrary objects. With that restriction, we should be free to use `compareTo()` in the definition of the generic class or method.

There are two different but related syntaxes for putting restrictions on the types that are used in generic programming. One of these is ***bounded type parameters***, which are used as formal type parameters in generic class and method definitions; a bounded type parameter would be used in place of the simple type parameter *T* in "`class GenericClass<T> ...`" or in "`public static <T> void genericMethod(...`". The second syntax is ***wildcard types***, which are used as type parameters in the declarations of variables and of formal parameters in method definitions; a wildcard type could be used in place of the type parameter *String* in the declaration statement "`List<String> list;`" or in the formal parameter list "`void max(Collection<String> c)`". We will look at wildcard types first, and we will return to the topic of bounded types later in this section.

Let's start with a simple example in which a wildcard type is useful. Suppose that *Shape* is a class that defines a method `public void draw()`, and suppose that *Shape* has subclasses such as *Rect* and *Oval*. Suppose that we want a method that can draw all the shapes in a collection of *Shapes*. We might try:

```
public static void drawAll(Collection<Shape> shapes) {
   for ( Shape s : shapes )
      s.draw();
}
```

This method works fine if we apply it to a variable of type *Collection<Shape>*, or *ArrayList<Shape>*, or any other collection class with type parameter *Shape*. Suppose, however, that you have a list of *Rects* stored in a variable named `rectangles` of type *Collection<Rect>*. Since *Rects* are *Shapes*, you might expect to be able to call `drawAll(rectangles)`. Unfortunately, this will not work; a collection of *Rects* is **not** considered to be a collection of *Shapes*! The variable `rectangles` cannot be assigned to the formal parameter `shapes`. The solution is to replace the type parameter "`Shape`" in the declaration of `shapes` with the wildcard type "`? extends Shape`":

```
public static void drawAll(Collection<? extends Shape> shapes) {
   for ( Shape s : shapes )
      s.draw();
}
```

The wildcard type "`? extends Shape`" means roughly "any type that is either equal to *Shape* or that is a subclass of *Shape*". When the parameter `shapes` is declared to be of type *Collec-*

tion<? extends Shape>, it becomes possible to call the `drawAll` method with an actual parameter of type *Collection<Rect>* since *Rect* is a subclass of *Shape* and therefore matches the wildcard. We could also pass actual parameters to `drawAll` of type *ArrayList<Rect>* or *Set<Oval>* or *List<Oval>*. And we can still pass variables of type *Collection<Shape>* or *ArrayList<Shape>*, since the class *Shape* itself matches "`? extends Shape`". We have greatly increased the usefulness of the method by using the wildcard type.

(Although it is not essential, you might be interested in knowing *why* Java does not allow a collection of *Rects* to be used as a collection of *Shapes*, even though every *Rect* is considered to be a *Shape*. Consider the rather silly but legal method that adds an oval to a list of shapes:

```
static void addOval(List<Shape> shapes, Oval oval) {
   shapes.add(oval);
}
```

Suppose that `rectangles` is of type *List<Rect>*. It's illegal to call `addOval(rectangles,oval)`, because of the rule that a list of *Rects* is not a list of *Shapes*. If we dropped that rule, then `addOval(rectangles,oval)` would be legal, and it would add an *Oval* to a list of *Rects*. This would be bad: Since *Oval* is not a subclass of *Rect*, an *Oval* is **not** a *Rect*, and a list of *Rects* should never be able to contain an *Oval*. The method call `addOval(rectangles,oval)` does not make sense and **should** be illegal, so the rule that a collection of *Rects* is not a collection of *Shapes* is a good rule.)

As another example, consider the method `addAll()` from the interface *Collection<T>*. In my description of this method in Subsection 10.1.4, I say that for a collection, `coll`, of type *Collection<T>*, `coll.addAll(coll2)` "adds all the objects in `coll2` to `coll`. The parameter, `coll2`, can be any collection of type *Collection<T>*. However, it can also be more general. For example, if *T* is a class and *S* is a sub-class of *T*, then `coll2` can be of type *Collection<S>*. This makes sense because any object of type *S* is automatically of type *T* and so can legally be added to `coll`." If you think for a moment, you'll see that what I'm describing here, a little awkwardly, is a use of wildcard types: We don't want to require `coll2` to be a collection of objects of type *T*; we want to allow collections of any subclass of *T*. To be more specific, let's look at how a similar `addAll()` method could be added to the generic *Queue* class that was defined earlier in this section:

```
class Queue<T> {
   private LinkedList<T> items = new LinkedList<T>();
   public void enqueue(T item) {
      items.addLast(item);
   }
   public T dequeue() {
      return items.removeFirst();
   }
   public boolean isEmpty() {
      return (items.size() == 0);
   }
   public void addAll(Collection<? extends T> collection) {
         // Add all the items from the collection to the end of the queue
      for ( T item : collection )
         enqueue(item);
   }
}
```

Here, *T* is a type parameter in the generic class definition. We are combining wildcard types with generic classes. Inside the generic class definition, "`T`" is used as if it is a specific, though unknown, type. The wildcard type "`? extends T`" means some type that extends that specific type. When we create a queue of type *Queue<Shape>*, "`T`" refers to "Shape", and the wildcard type "`? extends T`" in the class definition means "`? extends Shape`", meaning that the `addAll` method of the queue can be applied to collections of *Rects* and *Ovals* as well as to collections of *Shapes*.

The for-each loop in the definition of `addAll` iterates through the `collection` using a variable, `item`, of type *T*. Now, `collection` can be of type *Collection<S>*, where *S* is a subclass of *T*. Since `item` is of type *T*, not *S*, do we have a problem here? No, no problem. As long as *S* is a subclass of *T*, a value of type *S* can be assigned to a variable of type *T*. The restriction on the wildcard type makes everything work nicely.

The `addAll` method adds all the items from a collection to the queue. Suppose that we wanted to do the opposite: Add all the items that are currently on the queue to a given collection. An instance method defined as

```
public void addAllTo(Collection<T> collection)
```

would only work for collections whose base type is exactly the same as *T*. This is too restrictive. We need some sort of wildcard. However, "`? extends T`" won't work. Suppose we try it:

```
public void addAllTo(Collection<? extends T> collection) {
      // Remove all items currently on the queue and add them to collection
   while ( ! isEmpty() ) {
      T item = dequeue();  // Remove an item from the queue.
      collection.add( item );  // Add it to the collection.  ILLEGAL!!
   }
}
```

The problem is that we can't add an `item` of type *T* to a collection that might only be able to hold items belonging to some subclass, *S*, of *T*. The containment is going in the wrong direction: An `item` of type *T* is not necessarily of type *S*. For example, if we have a queue of type *Queue<Shape>*, it doesn't make sense to add items from the queue to a collection of type *Collection<Rect>*, since not every *Shape* is a *Rect*. On the other hand, if we have a *Queue<Rect>*, it would make sense to add items from that queue to a *Collection<Shape>* or indeed to any collection *Collection<S>* where *S* is a **super**class of *Rect*.

To express this type of relationship, we need a new kind of type wildcard: "`? super T`". This wildcard means, roughly, "either *T* itself or any class that is a superclass of *T*." For example, *Collection<? super Rect>* would match the types *Collection<Shape>*, *ArrayList<Object>*, and *Set<Rect>*. This is what we need for our `addAllTo` method. With this change, our complete generic queue class becomes:

```
class Queue<T> {
   private LinkedList<T> items = new LinkedList<T>();
   public void enqueue(T item) {
      items.addLast(item);
   }
   public T dequeue() {
      return items.removeFirst();
   }
   public boolean isEmpty() {
      return (items.size() == 0);
```

```
    }
    public void addAll(Collection<? extends T> collection) {
          // Add all the items from the collection to the end of the queue
       for ( T item : collection )
          enqueue(item);
    }
    public void addAllTo(Collection<? super T> collection) {
          // Remove all items currently on the queue and add them to collection
       while ( ! isEmpty() ) {
          T item = dequeue();  // Remove an item from the queue.
          collection.add( item );  // Add it to the collection.
       }
    }
}
```

In a wildcard type such as "`? extends T`", *T* can be an `interface` instead of a class. Note that the term "`extends`" (not "`implements`") is used in the wildcard type, even if *T* is an interface. For example, we will see that *Runnable* is an `interface` that defines the method `public void run()`. (Runnable objects are usually associated with threads; see Chapter 12.) Here is a method that runs all the objects in a collection of *Runnables* by executing the `run()` method from each runnable object:

```
public static runAll( Collection<? extends Runnable> runnables ) {
   for ( Runnable runnable : runnables ) {
      runnable.run();
   }
}
```

* * *

Wildcard types are used **only** as type parameters in parameterized types, such as *Collection<? extends Runnable>*. The place where a wildcard type is most likely to occur, by far, is in a formal parameter list, where the wildcard type is used in the declaration of the type of a formal parameter. However, they can also be used in a few other places. For example, they can be used in the type specification in a variable declaration statement.

One final remark: The wildcard type "`<?>`" is equivalent to "`<? extends Object>`". That is, it matches any possible type. For example, the `removeAll()` method in the generic interface *Collections<T>* is declared as

```
public boolean removeAll( Collection<?> c ) { ...
```

This just means that the `removeAll` method can be applied to any collection of any type of object.

10.5.4 Bounded Types

Wildcard types don't solve all of our problems. They allow us to generalize method definitions so that they can work with collections of objects of various types, rather than just a single type. However, they do not allow us to restrict the types that are allowed as type parameters in a generic class or method definition. Bounded types exist for this purpose.

We start with a small, not very realistic example. Suppose that you would like to create groups of GUI components using a generic class named *ComponentGroup*. For example, the parameterized type *ComponentGroup<JButton>* would represent a group of *JButtons*, while

ComponentGroup<JPanel> would represent a group of *JPanels*. The class will include methods that can be called to apply certain operations to all components in the group at once. For example, there will be an instance method of the form

```
public void repaintAll() {
   .
   .  // Call the repaint() method of every component in the group.
   .
}
```

The problem is that the `repaint()` method is defined in a *JComponent* object, but not for objects of arbitrary type. It wouldn't make sense to allow types such as *ComponentGroup<String>* or *ComponentGroup<Integer>*, since *Strings* and *Integers* don't have `repaint()` methods. We need some way to restrict the type parameter *T* in *ComponentGroup<T>* so that only *JComponent* and subclasses of *JComponent* are allowed as actual type parameters. We can do this by using the ***bounded type*** "`T extends JComponent`" instead of a plain "`T`" in the definition of the class:

```
public class ComponentGroup<T extends JComponent> {
   private ArrayList<T> components; // For storing the components in this group.
   public void repaintAll() {
      for ( JComponent c : components )
         if (c != null)
            c.repaint();
   }
   public void setAllEnabled( boolean enable ) {
      for ( JComponent c : components )
         if (c != null)
            c.setEnabled(enable);
      }
   }
   public void add( T c ) {  // Add a value c, of type T, to the group.
      components.add(c);
   }
   .
   .  // Additional methods and constructors.
   .
}
```

The restriction "`extends JComponent`" on `T` makes it illegal to create the parameterized types *ComponentGroup<String>* and *ComponentGroup<Integer>*, since the actual type parameter that replaces "`T`" is required to be either *JComponent* itself or a subclass of *JComponent*. With this restriction, we know—and, more important, the compiler knows—that the objects in the group are of type *JComponent* and the operations `c.repaint()` and `c.setEnabled()` are defined for any `c` in the group.

In general, a bounded type parameter "`T extends SomeType`" means roughly "a type, *T*, that is either equal to *SomeType* or is a subclass of *SomeType*", and the upshot is that any object of type *T* is also of type *SomeType*, and any operation that is defined for objects of type *SomeType* is defined for objects of type *T*. The type *SomeType* doesn't have to be the name of a class. It can be any name that represents an actual object type. For example, it can be an `interface` or even a parameterized type.

Bounded types and wildcard types are clearly related. They are, however, used in very different ways. A bounded type can be used only as a formal type parameter in the definition

of a generic method, class, or interface. A wildcard type is used most often to declare the type of a formal parameter in a method and cannot be used as a formal type parameter. One other difference, by the way, is that, in contrast to wildcard types, bounded type parameters can only use "`extends`", never "`super`".

Bounded type parameters can be used when declaring generic methods. For example, as an alternative to the generic *ComponentGroup* class, one could write a free-standing generic `static` method that can repaint any collection of *JComponents* as follows:

```
public static <T extends JComponent> void repaintAll(Collection<T> comps) {
   for ( JComponent c : comps )
      if (c != null)
         c.repaint();
}
```

Using "`<T extends JComponent>`" as the formal type parameter means that the method can only be called for collections whose base type is *JComponent* or some subclass of *JComponent*. Thus, it is legal to call `repaintAll(coll)` where `coll` is of type *List<JPanel>* but not where `coll` is of type *Set<String>*.

Note that we don't really need a generic type parameter in this case. We can write an equivalent method using a wildcard type:

```
public static void repaintAll(Collection<? extends JComponent> comps) {
   for ( JComponent c : comps )
      if (c != null)
         c.repaint();
}
```

In this situation, the version that uses the wildcard type is to be preferred, since the implementation is simpler. However, there are some situations where a generic method with a bounded type parameter cannot be rewritten using a wildcard type. Note that a generic type parameter gives a name, such as *T*, to the unknown type, while a wildcard type does not give a name to the unknown type. The name makes it possible to refer to the unknown type in the body of the method that is being defined. If a generic method definition uses the generic type name more than once or uses it outside the formal parameter list of the method, then the generic type cannot be replaced with a wildcard type.

Let's look at a generic method in which a bounded type parameter is essential. In Subsection 10.2.1, I presented a code segment for inserting a string into a sorted list of strings, in such a way that the modified list is still in sorted order. Here is the same code, but this time in the form of a method definition (and without the comments):

```
static void sortedInsert(List<String> sortedList, String newItem) {
   ListIterator<String> iter = sortedList.listIterator();
   while (iter.hasNext()) {
      String item = iter.next();
      if (newItem.compareTo(item) <= 0) {
         iter.previous();
         break;
      }
   }
   iter.add(newItem);
}
```

This method works fine for lists of strings, but it would be nice to have a generic method that can be applied to lists of other types of objects. The problem, of course, is that the code assumes that the `compareTo()` method is defined for objects in the list, so the method can only work for lists of objects that implement the *Comparable* interface. We can't simply use a wildcard type to enforce this restriction. Suppose we try to do it, by replacing *List<String>* with *List<? extends Comparable>*:

```
static void sortedInsert(List<? extends Comparable> sortedList, ???? newItem) {
   ListIterator<????> iter = stringList.listIterator();
   ...
```

We immediately run into a problem, because we have no name for the unknown type represented by the wildcard. We **need** a name for that type because the type of `newItem` and of `iter` should be the same as the type of the items in the list. The problem is solved if we write a generic method with a bounded type parameter, since then we have a name for the unknown type, and we can write a valid generic method:

```
static <T extends Comparable> void sortedInsert(List<T> sortedList, T newItem) {
   ListIterator<T> iter = sortedList.listIterator();
   while (iter.hasNext()) {
      T item = iter.next();
      if (newItem.compareTo(item) <= 0) {
         iter.previous();
         break;
      }
   }
   iter.add(newItem);
}
```

There is still one technicality to cover in this example. *Comparable* is itself a parameterized type, but I have used it here without a type parameter. This is legal but the compiler might give you a warning about using a "raw type." In fact, the objects in the list should implement the parameterized interface *Comparable<T>*, since they are being compared to items of type *T*. This just means that instead of using *Comparable* as the type bound, we should use *Comparable<T>*:

```
static <T extends Comparable<T>> void sortedInsert(List<T> sortedList, ...
```

* * *

With this example, I will leave the topic of generic types and generic programming. In this chapter, I have occasionally used terms such as "strange" and "weird" to talk about generic programming in Java. I will confess that I have some affection for the more simple-minded generic programming style of Smalltalk. Nevertheless, I recognize the power and increased robustness of generics in Java. I hope that I have convinced you that using the Java Collection Framework is reasonably natural and straightforward, and that using it can save you a lot of time and effort compared to repeatedly recoding the same data structures and algorithms from scratch. Things become more technical when you start writing new generic classes and methods of your own, and the syntax is (as I've said) a little strange. But with some practice, you'll get used to the syntax and will find that it's not that difficult after all.

Exercises for Chapter 10

1. Rewrite the *PhoneDirectory* class from Subsection 7.4.2 so that it uses a *TreeMap* to store directory entries, instead of an array. (Doing this was suggested in Subsection 10.3.1.)

2. In mathematics, several operations are defined on sets. The ***union*** of two sets A and B is a set that contains all the elements that are in A together with all the elements that are in B. The ***intersection*** of A and B is the set that contains elements that are in both A and B. The ***difference*** of A and B is the set that contains all the elements of A **except** for those elements that are also in B.

Suppose that `A` and `B` are variables of type `set` in Java. The mathematical operations on `A` and `B` can be computed using methods from the *Set* interface. In particular: `A.addAll(B)` computes the *union* of `A` and `B`; `A.retainAll(B)` computes the *intersection* of `A` and `B`; and `A.removeAll(B)` computes the *difference* of `A` and `B`. (These operations change the contents of the set `A`, while the mathematical operations create a new set without changing `A`, but that difference is not relevant to this exercise.)

For this exercise, you should write a program that can be used as a "set calculator" for simple operations on sets of non-negative integers. (Negative integers are not allowed.) A set of such integers will be represented as a list of integers, separated by commas and, optionally, spaces and enclosed in square brackets. For example: `[1,2,3]` or `[17, 42, 9, 53, 108]`. The characters `+`, `*`, and `-` will be used for the union, intersection, and difference operations. The user of the program will type in lines of input containing two sets, separated by an operator. The program should perform the operation and print the resulting set. Here are some examples:

```
 Input                                    Output
-------------------------              -------------------
 [1, 2, 3] + [3,  5,  7]                 [1, 2, 3, 5, 7]
 [10,9,8,7] * [2,4,6,8]                  [8]
 [ 5, 10, 15, 20 ] - [ 0, 10, 20 ]       [5, 15]
```

To represent sets of non-negative integers, use sets of type *TreeSet<Integer>*. Read the user's input, create two *TreeSets*, and use the appropriate *TreeSet* method to perform the requested operation on the two sets. Your program should be able to read and process any number of lines of input. If a line contains a syntax error, your program should not crash. It should report the error and move on to the next line of input. (Note: To print out a *Set*, `A`, of *Integers*, you can just say `System.out.println(A)`. I've chosen the syntax for sets to be the same as that used by the system for outputting a set.)

3. The fact that Java has a *HashMap* class means that no Java programmer has to write an implementation of hash tables from scratch—unless, of course, that programmer is a computer science student.

For this exercise, you should write a hash table in which both the keys and the values are of type *String*. (This is not an exercise in generic programming; do not try to write a generic class.) Write an implementation of hash tables from scratch. Define the following methods: `get(key)`, `put(key,value)`, `remove(key)`, `containsKey(key)`, and `size()`. Remember that every object, `obj`, has a method `obj.hashCode()` that can be used for computing a hash code for the object, so at least you don't have to define your own hash function. Do not use **any** of Java's built-in generic types; create your own linked lists

using nodes as covered in Subsection 9.2.2. However, you do **not** have to worry about increasing the size of the table when it becomes too full.

You should also write a short program to test your solution.

4. A ***predicate*** is a boolean-valued function with one parameter. Some languages use predicates in generic programming. Java doesn't, but this exercise looks at how predicates might work in Java.

In Java, we could implement "predicate objects" by defining a generic interface:

```
public interface Predicate<T> {
    public boolean test( T obj );
}
```

The idea is that an object that implements this interface knows how to "test" objects of type *T* in some way. Define a class that contains the following generic static methods for working with predicate objects. The name of the class should be *Predicates*, in analogy with the standard class *Collections* that provides various `static` methods for working with collections.

```
public static <T> void remove(Collection<T> coll, Predicate<T> pred)
   // Remove every object, obj, from coll for which
   // pred.test(obj) is true.

public static <T> void retain(Collection<T> coll, Predicate<T> pred)
   // Remove every object, obj, from coll for which
   // pred.test(obj) is false.  (That is, retain the
   // objects for which the predicate is true.)

public static <T> List<T> collect(Collection<T> coll, Predicate<T> pred)
   // Return a List that contains all the objects, obj,
   // from the collection, coll, such that pred.test(obj)
   // is true.

public static <T> int find(ArrayList<T> list, Predicate<T> pred)
   // Return the index of the first item in list
   // for which the predicate is true, if any.
   // If there is no such item, return -1.
```

(In C++, methods similar to these are included as a standard part of the generic programming framework.)

5. An example in Subsection 10.4.2 concerns the problem of making an index for a book. A related problem is making a ***concordance*** for a document. A concordance lists every word that occurs in the document, and for each word it gives the line number of every line in the document where the word occurs. All the subroutines for creating an index that were presented in Subsection 10.4.2 can also be used to create a concordance. The only real difference is that the integers in a concordance are line numbers rather than page numbers.

Write a program that can create a concordance. The document should be read from an input file, and the concordance data should be written to an output file. You can use the indexing subroutines from Subsection 10.4.2, modified to write the data to `TextIO` instead of to `System.out`. (You will need to make these subroutines `static`.) The input and output files should be selected by the user when the program is run. The sample

program *WordCount.java*, from Subsection 10.4.4, can be used as a model of how to use files. That program also has a useful subroutine that reads one word from input.

As you read the file, you want to take each word that you encounter and add it to the concordance along with the current line number. Keeping track of the line numbers is one of the trickiest parts of the problem. In an input file, the end of each line in the file is marked by the newline character, '\n'. Every time you encounter this character, you have to add one to the line number. WordCount.java ignores ends of lines. Because you need to find and count the end-of-line characters, your program cannot process the input file in exactly the same way as does WordCount.java. Also, you will need to detect the end of the file. The function `TextIO.peek()`, which is used to look ahead at the next character in the input, returns the value `TextIO.EOF` at end-of-file, after all the characters in the file have been read.

Because it is so common, don't include the word "the" in your concordance. Also, do not include words that have length less than 3.

6. The sample program *SimpleInterpreter.java* from Subsection 10.4.1 can carry out commands of the form "let variable = expression" or "print expression". That program can handle expressions that contain variables, numbers, operators, and parentheses. Extend the program so that it can also handle the standard mathematical functions `sin`, `cos`, `tan`, `abs`, `sqrt`, and `log`. For example, the program should be able to evaluate an expression such as `sin(3*x-7)+log(sqrt(y))`, assuming that the variables `x` and `y` have been given values. Note that the name of a function must be followed by an expression that is enclosed in parentheses.

In the original program, a symbol table holds a value for each variable that has been defined. In your program, you should add another type of symbol to the table to represent standard functions. You can use the following nested enumerated type and class for this purpose:

```
private enum Functions { SIN, COS, TAN, ABS, SQRT, LOG }

/**
 * An object of this class represents one of the standard functions.
 */
private static class StandardFunction {

   /**
    * Tells which function this is.
    */
   Functions functionCode;

   /**
    * Constructor creates an object to represent one of
    * the standard functions
    * @param code which function is represented.
    */
   StandardFunction(Functions code) {
      functionCode = code;
   }

   /**
    * Finds the value of this function for the specified
    * parameter value, x.
```

```
     */
    double evaluate(double x) {
       switch(functionCode) {
       case SIN:
          return Math.sin(x);
       case COS:
          return Math.cos(x);
       case TAN:
          return Math.tan(x);
       case ABS:
          return Math.abs(x);
       case SQRT:
          return Math.sqrt(x);
       default:
          return Math.log(x);
       }
    }

} // end class StandardFunction
```

Add a symbol to the symbol table to represent each function. The key is the name of the function and the value is an object of type *StandardFunction* that represents the function. For example:

```
symbolTable.put("sin", new StandardFunction(StandardFunction.SIN));
```

In SimpleInterpreter.java, the symbol table is a map of type *HashMap<String,Double>*. It's not legal to use a *StandardFunction* as the value in such a map, so you will have to change the type of the map. The map has to hold two different types of objects. The easy way to make this possible is to create a map of type *HashMap<String,Object>*. (A better way is to create a general type to represent objects that can be values in the symbol table, and to define two subclasses of that class, one to represent variables and one to represent standard functions, but for this exercise, you should do it the easy way.)

In your parser, when you encounter a word, you have to be able to tell whether it's a variable or a standard function. Look up the word in the symbol table. If the associated object is non-null and is of type *Double*, then the word is a variable. If it is of type *StandardFunction*, then the word is a function. Remember that you can test the type of an object using the `instanceof` operator. For example: `if (obj instanceof Double)`

Quiz on Chapter 10

1. What is meant by *generic programming* and what is the alternative?

2. Why can't you make an object of type *LinkedList<int>*? What should you do instead?

3. What is an *iterator* and why are iterators necessary for generic programming?

4. Suppose that `integers` is a variable of type *Collection<Integer>*. Write a code segment that uses an iterator to compute the sum of all the integer values in the collection. Write a second code segment that does the same thing using a for-each loop.

5. Interfaces such as *List*, *Set*, and *Map* define *abstract data types.* Explain what this means.

6. What is the fundamental property that distinguishes *Sets* from other types of *Collections*?

7. What is the essential difference in functionality between a *TreeMap* and a *HashMap*?

8. Explain what is meant by a *hash code.*

9. Modify the following *Date* class so that it implements the interface *Comparable<Date>*. The ordering on objects of type *Date* should be the natural, chronological ordering.

```
class Date {
   int month;  // Month number in range 1 to 12.
   int day;    // Day number in range 1 to 31.
   int year;   // Year number.
   Date(int m, int d, int y) {
      month = m;
      day = d;
      year = y;
   }
}
```

10. Suppose that `syllabus` is a variable of type *TreeMap<Date,String>*, where *Date* is the class from the preceding exercise. Write a code segment that will write out the value string for every key that is in the month of December, 2010.

11. Write a generic class *Stack<T>* that can be used to represent stacks of objects of type *T*. The class should include methods `push()`, `pop()`, and `isEmpty()`. Inside the class, use an *ArrayList* to hold the items on the stack.

12. Write a generic method, using a generic type parameter `<T>`, that replaces every occurrence in an *ArrayList<T>* of a specified item with a specified replacement item. The list and the two items are parameters to the method. Both items are of type *T*. Take into account the fact that the item that is being replaced might be `null`. For a non-null item, use `equals()` to do the comparison.

Chapter 11

Advanced Input/Output: Streams, Files, and Networking

Computer programs are only useful if they interact with the rest of the world in some way. This interaction is referred to as ***input/output***, or ***I/O***. Up until now, this book has concentrated on just one type of interaction: interaction with the user, through either a graphical user interface or a command-line interface. But the user is only one possible source of information and only one possible destination for information. We have already encountered one other type of input/output, since *TextIO* can read data from files and write data to files. However, Java has an input/output framework that provides much more power and flexibility than does *TextIO*, and that covers other kinds of I/O in addition to files. Most importantly, it supports communication over network connections. In Java, input/output involving files and networks is based on ***streams***, which are objects that support I/O commands that are similar to those that you have already used. In fact, standard output (`System.out`) and standard input (`System.in`) are examples of streams.

Working with files and networks requires familiarity with exceptions, which were covered in Chapter 8. Many of the subroutines that are used can throw exceptions that require mandatory exception handling. This generally means calling the subroutine in a `try..catch` statement that can deal with the exception if one occurs. Effective network communication also requires the use of threads, which will be covered in Chapter 12. We will look at the basic networking API in this chapter, but we will return to the topic of threads and networking in Section 12.4.

11.1 Streams, Readers, and Writers

Without the ability to interact with the rest of the world, a program would be useless. The interaction of a program with the rest of the world is referred to as ***input/output*** or I/O. Historically, one of the hardest parts of programming language design has been coming up with good facilities for doing input and output. A computer can be connected to many different types of input and output devices. If a programming language had to deal with each type of device as a special case, the complexity would be overwhelming. One of the major achievements in the history of programming has been to come up with good abstractions for representing I/O devices. In Java, the main I/O abstractions are called ***streams***. Other I/O abstractions, such as "files" and "channels" also exist, but in this section we will look only at streams. Every stream represents either a source of input or a destination to which output can be sent.

11.1.1 Character and Byte Streams

When dealing with input/output, you have to keep in mind that there are two broad categories of data: machine-formatted data and human-readable text. Machine-formatted data is represented in binary form, the same way that data is represented inside the computer, that is, as strings of zeros and ones. Human-readable data is in the form of characters. When you read a number such as `3.141592654`, you are reading a sequence of characters and interpreting them as a number. The same number would be represented in the computer as a bit-string that you would find unrecognizable.

To deal with the two broad categories of data representation, Java has two broad categories of streams: ***byte streams*** for machine-formatted data and ***character streams*** for human-readable data. There are many predefined classes that represent streams of each type.

An object that **outputs** data to a byte stream belongs to one of the subclasses of the abstract class *OutputStream*. Objects that **read** data from a byte stream belong to subclasses of *InputStream*. If you write numbers to an *OutputStream*, you won't be able to read the resulting data yourself. But the data can be read back into the computer with an *InputStream*. The writing and reading of the data will be very efficient, since there is no translation involved: the bits that are used to represent the data inside the computer are simply copied to and from the streams.

For reading and writing human-readable character data, the main classes are the abstract classes *Reader* and *Writer*. All character stream classes are subclasses of one of these. If a number is to be written to a *Writer* stream, the computer must translate it into a human-readable sequence of characters that represents that number. Reading a number from a *Reader* stream into a numeric variable also involves a translation, from a character sequence into the appropriate bit string. (Even if the data you are working with consists of characters in the first place, such as words from a text editor, there might still be some translation. Characters are stored in the computer as 16-bit Unicode values. For people who use Western alphabets, character data is generally stored in files in ASCII code, which uses only 8 bits per character. The *Reader* and *Writer* classes take care of this translation, and can also handle non-western alphabets in countries that use them.)

Byte streams can be useful for direct machine-to-machine communication, and they can sometimes be useful for storing data in files, especially when large amounts of data need to be stored efficiently, such as in large databases. However, binary data is *fragile* in the sense that its meaning is not self-evident. When faced with a long series of zeros and ones, you have to know what information it is meant to represent and how that information is encoded before you will be able to interpret it. Of course, the same is true to some extent for character data, which is itself coded into binary form. But the binary encoding of character data has been standardized and is well understood, and data expressed in character form can be made meaningful to human readers. The current trend seems to be towards increased use of character data, represented in a way that will make its meaning as self-evident as possible. We'll look at one way this is done in Section 11.5.

I should note that the original version of Java did not have character streams, and that for ASCII-encoded character data, byte streams are largely interchangeable with character streams. In fact, the standard input and output streams, `System.in` and `System.out`, are byte streams rather than character streams. However, you should use *Readers* and *Writers* rather than *InputStreams* and *OutputStreams* when working with character data, even when working with the standard ASCII character set.

The standard stream classes discussed in this section are defined in the package `java.io`,

along with several supporting classes. You must `import` the classes from this package if you want to use them in your program. That means either importing individual classes or putting the directive "`import java.io.*;`" at the beginning of your source file. Streams are necessary for working with files and for doing communication over a network. They can also be used for communication between two concurrently running threads, and there are stream classes for reading and writing data stored in the computer's memory.

The beauty of the stream abstraction is that it is as easy to write data to a file or to send data over a network as it is to print information on the screen.

* * *

The basic I/O classes *Reader*, *Writer*, *InputStream*, and *OutputStream* provide only very primitive I/O operations. For example, the *InputStream* class declares the instance method

```
public int read() throws IOException
```

for reading one byte of data, as a number in the range 0 to 255, from an input stream. If the end of the input stream is encountered, the `read()` method will return the value -1 instead. If some error occurs during the input attempt, an exception of type *IOException* is thrown. Since *IOException* is an exception class that requires mandatory exception-handling, this means that you can't use the `read()` method except inside a `try` statement or in a subroutine that is itself declared with a "`throws IOException`" clause. (Mandatory exception handling was covered in Subsection 8.3.3.)

The *InputStream* class also defines methods for reading multiple bytes of data in one step into an array of `bytes`. However, *InputStream* provides no convenient methods for reading other types of data, such as **int** or **double**, from a stream. This is not a problem because you'll never use an object of type *InputStream* itself. Instead, you'll use subclasses of *InputStream* that add more convenient input methods to *InputStream's* rather primitive capabilities. Similarly, the *OutputStream* class defines a primitive output method for writing one byte of data to an output stream. The method is defined as:

```
public void write(int b) throws IOException
```

The parameter is of type **int** rather than **byte**, but the parameter value is type-cast to type **byte** before it is written; this effectively discards all but the eight low order bits of `b`. Again, in practice, you will almost always use higher-level output operations defined in some subclass of *OutputStream*.

The *Reader* and *Writer* classes provide the analogous low-level `read` and `write` methods. As in the byte stream classes, the parameter of the `write(c)` method in *Writer* and the return value of the `read()` method in *Reader* are of type **int**, but in these character-oriented classes, the I/O operations read and write characters rather than bytes. The return value of `read()` is `-1` if the end of the input stream has been reached. Otherwise, the return value must be type-cast to type **char** to obtain the character that was read. In practice, you will ordinarily use higher level I/O operations provided by sub-classes of *Reader* and *Writer*, as discussed below.

11.1.2 PrintWriter

One of the neat things about Java's I/O package is that it lets you add capabilities to a stream by "wrapping" it in another stream object that provides those capabilities. The wrapper object is also a stream, so you can read from or write to it—but you can do so using fancier operations than those available for basic streams.

For example, *PrintWriter* is a subclass of *Writer* that provides convenient methods for outputting human-readable character representations of all of Java's basic data types. If you have an object belonging to the *Writer* class, or any of its subclasses, and you would like to use *PrintWriter* methods to output data to that *Writer*, all you have to do is wrap the *Writer* in a *PrintWriter* object. You do this by constructing a new *PrintWriter* object, using the *Writer* as input to the constructor. For example, if `charSink` is of type *Writer*, then you could say

```
PrintWriter printableCharSink = new PrintWriter(charSink);
```

When you output data to `printableCharSink`, using the high-level output methods in *PrintWriter*, that data will go to exactly the same place as data written directly to `charSink`. You've just provided a better interface to the same output stream. For example, this allows you to use *PrintWriter* methods to send data to a file or over a network connection.

For the record, if `out` is a variable of type *PrintWriter*, then the following methods are defined:

- `out.print(x)` — prints the value of `x`, represented in the form of a string of characters, to the output stream; `x` can be an expression of any type, including both primitive types and object types. An object is converted to string form using its `toString()` method. A `null` value is represented by the string "null".
- `out.println()` — outputs an end-of-line to the output stream.
- `out.println(x)` — outputs the value of `x`, followed by an end-of-line; this is equivalent to `out.print(x)` followed by `out.println()`.
- `out.printf(formatString, x1, x2, ...)` — does formatted output of `x1`, `x2`, `...` to the output stream. The first parameter is a string that specifies the format of the output. There can be any number of additional parameters, of any type, but the types of the parameters must match the formatting directives in the format string. Formatted output for the standard output stream, `System.out`, was introduced in Subsection 2.4.4, and `out.printf` has the same functionality.
- `out.flush()` — ensures that characters that have been written with the above methods are actually sent to the output destination. In some cases, notably when writing to a file or to the network, it might be necessary to call this method to force the output to actually appear at the destination.

Note that none of these methods will ever throw an *IOException*. Instead, the *PrintWriter* class includes the method

```
public boolean checkError()
```

which will return `true` if any error has been encountered while writing to the stream. The *PrintWriter* class catches any *IOExceptions* internally, and sets the value of an internal error flag if one occurs. The `checkError()` method can be used to check the error flag. This allows you to use *PrintWriter* methods without worrying about catching exceptions. On the other hand, to write a fully robust program, you should call `checkError()` to test for possible errors whenever you use a *PrintWriter*.

11.1.3 Data Streams

When you use a *PrintWriter* to output data to a stream, the data is converted into the sequence of characters that represents the data in human-readable form. Suppose you want to output

the data in byte-oriented, machine-formatted form? The `java.io` package includes a byte-stream class, *DataOutputStream* that can be used for writing data values to streams in internal, binary-number format. *DataOutputStream* bears the same relationship to *OutputStream* that *PrintWriter* bears to *Writer*. That is, whereas *OutputStream* only has methods for outputting bytes, *DataOutputStream* has methods `writeDouble(double x)` for outputting values of type **double**, `writeInt(int x)` for outputting values of type **int**, and so on. Furthermore, you can wrap any *OutputStream* in a *DataOutputStream* so that you can use the higher level output methods on it. For example, if `byteSink` is of type *OutputStream*, you could say

```
DataOutputStream dataSink = new DataOutputStream(byteSink);
```

to wrap `byteSink` in a *DataOutputStream*, `dataSink`.

For input of machine-readable data, such as that created by writing to a *DataOutputStream*, `java.io` provides the class *DataInputStream*. You can wrap any *InputStream* in a *DataInputStream* object to provide it with the ability to read data of various types from the byte-stream. The methods in the *DataInputStream* for reading binary data are called `readDouble()`, `readInt()`, and so on. Data written by a *DataOutputStream* is guaranteed to be in a format that can be read by a *DataInputStream*. This is true even if the data stream is created on one type of computer and read on another type of computer. The cross-platform compatibility of binary data is a major aspect of Java's platform independence.

In some circumstances, you might need to read character data from an *InputStream* or write character data to an *OutputStream*. This is not a problem, since characters, like all data, are represented as binary numbers. However, for character data, it is convenient to use *Reader* and *Writer* instead of *InputStream* and *OutputStream*. To make this possible, you can **wrap** a byte stream in a character stream. If `byteSource` is a variable of type *InputStream* and `byteSink` is of type *OutputStream*, then the statements

```
Reader charSource = new InputStreamReader( byteSource );
Writer charSink   = new OutputStreamWriter( byteSink );
```

create character streams that can be used to read character data from and write character data to the byte streams. In particular, the standard input stream `System.in`, which is of type *InputStream* for historical reasons, can be wrapped in a *Reader* to make it easier to read character data from standard input:

```
Reader charIn = new InputStreamReader( System.in );
```

As another application, the input and output streams that are associated with a network connection are byte streams rather than character streams, but the byte streams can be wrapped in character streams to make it easy to send and receive character data over the network. We will encounter network I/O in Section 11.4.

There are various ways for characters to be encoded as binary data. A particular encoding is known as a ***charset*** or ***character set***. Charsets have standardized names such as "UTF-16," "UTF-8," and "ISO-8859-1." In UTF-16, characters are encoded as 16-bit UNICODE values; this is the character set that is used internally by Java. UTF-8 is a way of encoding UNICODE characters using 8 bits for common ASCII characters and longer codes for other characters. ISO-8859-1, also known as "Latin-1," is an 8-bit encoding that includes ASCII characters as well as certain accented characters that are used in several European languages. *Readers* and *Writers* use the default charset for the computer on which they are running, unless you specify a different one. This can be done, for example, in a constructor such as

```
Writer charSink = new OutputStreamWriter( byteSink, "ISO-8859-1" );
```

Certainly, the existence of a variety of charset encodings has made text processing more complicated—unfortunate for us English-speakers but essential for people who use non-Western character sets. Ordinarily, you don't have to worry about this, but it's a good idea to be aware that different charsets exist in case you run into textual data encoded in a non-default way.

11.1.4 Reading Text

Much I/O is done in the form of human-readable characters. In view of this, it is surprising that Java does **not** provide a standard character input class that can read character data in a manner that is reasonably symmetrical with the character output capabilities of *PrintWriter*. (The *Scanner* class, introduced briefly in Subsection 2.4.6 and covered in more detail in Subsection 11.1.5, comes pretty close.) There is one basic case that is easily handled by a standard class. The *BufferedReader* class has a method

```
public String readLine() throws IOException
```

that reads one line of text from its input source. If the end of the stream has been reached, the return value is `null`. When a line of text is read, the end-of-line marker is read from the input stream, but it is not part of the string that is returned. Different input streams use different characters as end-of-line markers, but the `readLine` method can deal with all the common cases. (Traditionally, Unix computers, including Linux and Mac OS X, use a line feed character, `'\n'`, to mark an end of line; classic Macintosh used a carriage return character, `'\r'`; and Windows uses the two-character sequence "`\r\n`". In general, modern computers can deal correctly with all of these possibilities.)

Line-by-line processing is very common. Any *Reader* can be wrapped in a *BufferedReader* to make it easy to read full lines of text. If `reader` is of type *Reader*, then a *BufferedReader* wrapper can be created for `reader` with

```
BufferedReader in = new BufferedReader( reader );
```

This can be combined with the *InputStreamReader* class that was mentioned above to read lines of text from an *InputStream*. For example, we can apply this to `System.in`:

```
BufferedReader in;  // BufferedReader for reading from standard input.
in = new BufferedReader( new InputStreamReader( System.in ) );
try {
   String line = in.readLine();
   while ( line != null ) {
      processOneLineOfInput( line );
      line = in.readLine();
   }
}
catch (IOException e) {
}
```

This code segment reads and processes lines from standard input until an end-of-stream is encountered. (An end-of-stream is possible even for interactive input. For example, on at least some computers, typing a `Control-D` generates an end-of-stream on the standard input stream.) The `try..catch` statement is necessary because the `readLine` method can throw an exception of type *IOException*, which requires mandatory exception handling; an alternative to `try..catch` would be to declare that the method that contains the code "`throws IOException`". Also, remember that *BufferedReader*, *InputStreamReader*, and *IOException* must be imported from the package `java.io`.

* * *

Previously in this book, we have used the non-standard class *TextIO* for input both from users and from files. The advantage of *TextIO* is that it makes it fairly easy to read data values of any of the primitive types. Disadvantages include the fact that *TextIO* can only read from one file at a time, that it can't do I/O operations on network connections, and that it does not follow the same pattern as Java's built-in input/output classes.

I have written a class named *TextReader* to fix some of these disadvantages, while providing input capabilities similar to those of *TextIO*. Like *TextIO*, *TextReader* is a non-standard class, so you have to be careful to make it available to any program that uses it. The source code for the class can be found in the file *TextReader.java*.

Just as for many of Java's stream classes, an object of type *TextReader* can be used as a wrapper for an existing input stream, which becomes the source of the characters that will be read by the *TextReader*. (Unlike the standard classes, however, a *TextReader* is not itself a stream and cannot be wrapped inside other stream classes.) The constructors

```
public TextReader(Reader characterSource)
```

and

```
public TextReader(InputStream byteSource)
```

create objects that can be used to read human-readable data from the given *Reader* or *InputStream* using the convenient input methods of the *TextReader* class. In *TextIO*, the input methods were static members of the class. The input methods in the *TextReader* class are instance methods. The instance methods in a *TextReader* object read from the data source that was specified in the object's constructor. This makes it possible for several *TextReader* objects to exist at the same time, reading from different streams; those objects can then be used to read data from several files or other input sources at the same time.

A *TextReader* object has essentially the same set of input methods as the *TextIO* class. One big difference is how errors are handled. When a *TextReader* encounters an error in the input, it throws an exception of type *IOException*. This follows the standard pattern that is used by Java's standard input streams. *IOExceptions* require mandatory exception handling, so *TextReader* methods are generally called inside `try..catch` statements. If an *IOException* is thrown by the input stream that is wrapped inside a *TextReader*, that *IOException* is simply passed along. However, other types of errors can also occur. One such possible error is an attempt to read data from the input stream when there is no more data left in the stream. A *TextReader* throws an exception of type *TextReader.EndOfStreamException* when this happens. The exception class in this case is a nested class in the *TextReader* class; it is a subclass of *IOException*, so a `try..catch` statement that handles *IOExceptions* will also handle end-of-stream exceptions. However, having a class to represent end-of-stream errors makes it possible to detect such errors and provide special handling for them. Another type of error occurs when a *TextReader* tries to read a data value of a certain type, and the next item in the input stream is not of the correct type. In this case, the *TextReader* throws an exception of type *TextReader.BadDataException*, which is another subclass of *IOException*.

For reference, here is a list of some of the more useful instance methods in the *TextReader* class. All of these methods can throw exceptions of type *IOException*:

- `public char peek()` — looks ahead at the next character in the input stream, and returns that character. The character is not removed from the stream. If the next character is an end-of-line, the return value is '`\n`'. It is legal to call this method even if there is no more data left in the stream; in that case, the return value is the constant `TextReader.EOF`.

("EOF" stands for "End-Of-File," a term that is more commonly used than "End-Of-Stream", even though not all streams are files.)

- `public boolean eoln()` and `public boolean eof()` — convenience methods for testing whether the next thing in the file is an end-of-line or an end-of-file. Note that these methods do **not** skip whitespace. If `eof()` is false, you know that there is still at least one character to be read, but there might not be any more **non-blank** characters in the stream.
- `public void skipBlanks()` and `public void skipWhiteSpace()` — skip past whitespace characters in the input stream; `skipWhiteSpace()` skips all whitespace characters, including end-of-line while `skipBlanks()` only skips spaces and tabs.
- `public String getln()` — reads characters up to the next end-of-line (or end-of-stream), and returns those characters in a string. The end-of-line marker is read but is not part of the returned string. This will throw an exception if there are no more characters in the stream.
- `public char getAnyChar()` — reads and returns the next character from the stream. The character can be a whitespace character such as a blank or end-of-line. If this method is called after all the characters in the stream have been read, an exception is thrown.
- `public int getlnInt()`, `public double getlnDouble()`, `public char getlnChar()`, etc. — skip any whitespace characters in the stream, including end-of-lines, then read a value of the specified type, which will be the return value of the method. Any remaining characters on the line are then discarded, including the end-of-line marker. There is a method for each primitive type. An exception occurs if it's not possible to read a data value of the requested type.
- `public int getInt()`, `public double getDouble()`, `public char getChar()`, etc. — skip any whitespace characters in the stream, including end-of-lines, then read and return a value of the specified type. Extra characters on the line are **not** discarded and are still available to be read by subsequent input methods. There is a method for each primitive type. An exception occurs if it's not possible to read a data value of the requested type.

11.1.5 The Scanner Class

Since its introduction, Java has been notable for its lack of built-in support for basic input, and for its reliance on fairly advanced techniques for the support that it does offer. (This is my opinion, at least.) The *Scanner* class was introduced in Java 5.0 to make it easier to read basic data types from a character input source. It does not (again, in my opinion) solve the problem completely, but it is a big improvement. The *Scanner* class is in the package `java.util`.

Input routines are defined as instance methods in the *Scanner* class, so to use the class, you need to create a *Scanner* object. The constructor specifies the source of the characters that the *Scanner* will read. The scanner acts as a wrapper for the input source. The source can be a *Reader*, an *InputStream*, a *String*, or a *File*. (If a *String* is used as the input source, the *Scanner* will simply read the characters in the string from beginning to end, in the same way that it would process the same sequence of characters from a stream. The *File* class will be covered in the next section.) For example, you can use a *Scanner* to read from standard input by saying:

```
Scanner standardInputScanner = new Scanner( System.in );
```

and if `charSource` is of type *Reader*, you can create a *Scanner* for reading from `charSource` with:

```
Scanner scanner = new Scanner( charSource );
```

When processing input, a scanner usually works with ***tokens***. A token is a meaningful string of characters that cannot, for the purposes at hand, be further broken down into smaller meaningful pieces. A token can, for example, be an individual word or a string of characters that represents a value of type **double**. In the case of a scanner, tokens must be separated by "delimiters." By default, the delimiters are whitespace characters such as spaces and end-of-line markers, but you can change a *Scanner's* delimiters if you need to. In normal processing, whitespace characters serve simply to separate tokens and are discarded by the scanner. A scanner has instance methods for reading tokens of various types. Suppose that `scanner` is an object of type *Scanner*. Then we have:

- `scanner.next()` — reads the next token from the input source and returns it as a *String*.
- `scanner.nextInt()`, `scanner.nextDouble()`, and so on — reads the next token from the input source and tries to convert it to a value of type **int**, **double**, and so on. There are methods for reading values of any of the primitive types.
- `scanner.nextLine()` — reads an entire line from the input source, up to the next end-of-line and returns the line as a value of type *String*. The end-of-line marker is read but is not part of the return value. Note that this method is **not** based on tokens. An entire line is read and returned, including any whitespace characters in the line.

All of these methods can generate exceptions. If an attempt is made to read past the end of input, an exception of type *NoSuchElementException* is thrown. Methods such as `scanner.getInt()` will throw an exception of type *InputMismatchException* if the next token in the input does not represent a value of the requested type. The exceptions that can be generated do not require mandatory exception handling.

The *Scanner* class has very nice look-ahead capabilities. You can query a scanner to determine whether more tokens are available and whether the next token is of a given type. If `scanner` is of type *Scanner*:

- `scanner.hasNext()` — returns a **boolean** value that is true if there is at least one more token in the input source.
- `scanner.hasNextInt()`, `scanner.hasNextDouble()`, and so on — returns a **boolean** value that is true if there is at least one more token in the input source and that token represents a value of the requested type.
- `scanner.hasNextLine()` — returns a **boolean** value that is true if there is at least one more line in the input source.

Although the insistence on defining tokens only in terms of delimiters limits the usability of scanners to some extent, they are easy to use and are suitable for many applications. With so many input classes available—*BufferedReader*, *TextReader*, *Scanner*—you might have trouble deciding which one to use! In general, I would recommend using a *Scanner* unless you have some particular reason for preferring the *TextIO*-style input routines of *TextReader*. *BufferedReader* can be used as a lightweight alternative when all that you want to do is read entire lines of text from the input source.

11.1.6 Serialized Object I/O

The classes *PrintWriter*, *TextReader*, *Scanner*, *DataInputStream*, and *DataOutputStream* allow you to easily input and output all of Java's primitive data types. But what happens when you

want to read and write **objects**? Traditionally, you would have to come up with some way of encoding your object as a sequence of data values belonging to the primitive types, which can then be output as bytes or characters. This is called ***serializing*** the object. On input, you have to read the serialized data and somehow reconstitute a copy of the original object. For complex objects, this can all be a major chore. However, you can get Java to do all the work for you by using the classes *ObjectInputStream* and *ObjectOutputStream*. These are subclasses of *InputStream* and *OutputStream* that can be used for writing and reading serialized objects.

ObjectInputStream and *ObjectOutputStream* are wrapper classes that can be wrapped around arbitrary *InputStreams* and *OutputStreams*. This makes it possible to do object input and output on any byte stream. The methods for object I/O are `readObject()`, in *ObjectInputStream*, and `writeObject(Object obj)`, in *ObjectOutputStream*. Both of these methods can throw *IOExceptions*. Note that `readObject()` returns a value of type *Object*, which generally has to be type-cast to the actual type of the object that was read.

ObjectOutputStream also has methods `writeInt()`, `writeDouble()`, and so on, for outputting primitive type values to the stream, and *ObjectInputStream* has corresponding methods for reading primitive type values. These primitive type values can be interspersed with objects in the data.

Object streams are byte streams. The objects are represented in binary, machine-readable form. This is good for efficiency, but it does suffer from the fragility that is often seen in binary data. They suffer from the additional problem that the binary format of Java objects is very specific to Java, so the data in object streams is not easily available to programs written in other programming languages. For these reasons, object streams are appropriate mostly for short-term storage of objects and for transmitting objects over a network connection from one Java program to another. For long-term storage and for communication with non-Java programs, other approaches to object serialization are usually better. (See Subsection 11.5.2 for a character-based approach.)

ObjectInputStream and *ObjectOutputStream* only work with objects that implement an interface named *Serializable*. Furthermore, all of the instance variables in the object must be serializable. However, there is little work involved in making an object serializable, since the *Serializable* interface does not declare any methods. It exists only as a marker for the compiler, to tell it that the object is meant to be writable and readable. You only need to add the words "`implements Serializable`" to your class definitions. Many of Java's standard classes are already declared to be serializable, including all the component classes and many other classes in Swing and in the AWT. One of the programming examples in Section 11.3 uses object IO.

One warning about using *ObjectOutputStreams*: These streams are optimized to avoid writing the same object more than once. When an object is encountered for a second time, only a reference to the first occurrence is written. Unfortunately, if the object has been modified in the meantime, the new data will not be written. Because of this, *ObjectOutputStreams* are meant mainly for use with "immutable" objects that can't be changed after they are created. (*Strings* are an example of this.) However, if you do need to write mutable objects to an *ObjectOutputStream*, you can ensure that the full, correct version of the object can be written by calling the stream's `reset()` method before writing the object to the stream.

11.2 Files

The data and programs in a computer's main memory survive only as long as the power is on. For more permanent storage, computers use ***files***, which are collections of data stored on

a hard disk, on a USB memory stick, on a CD-ROM, or on some other type of storage device. Files are organized into ***directories*** (sometimes called ***folders***). A directory can hold other directories, as well as files. Both directories and files have names that are used to identify them.

Programs can read data from existing files. They can create new files and can write data to files. In Java, such input and output can be done using streams. Human-readable character data is read from a file using an object belonging to the class *FileReader*, which is a subclass of *Reader*. Similarly, data is written to a file in human-readable format through an object of type *FileWriter*, a subclass of *Writer*. For files that store data in machine format, the appropriate I/O classes are *FileInputStream* and *FileOutputStream*. In this section, I will only discuss character-oriented file I/O using the *FileReader* and *FileWriter* classes. However, *FileInputStream* and *FileOutputStream* are used in an exactly parallel fashion. All these classes are defined in the `java.io` package.

It's worth noting right at the start that applets which are downloaded over a network connection are not ordinarily allowed to access files. This is a security consideration. You can download and run an applet just by visiting a Web page with your browser. If downloaded applets had access to the files on your computer, it would be easy to write an applet that would destroy all the data on any computer that downloads it. To prevent such possibilities, there are a number of things that downloaded applets are not allowed to do. Accessing files is one of those forbidden things. Standalone programs written in Java, however, have the same access to your files as any other program. When you write a standalone Java application, you can use all the file operations described in this section.

11.2.1 Reading and Writing Files

The *FileReader* class has a constructor which takes the name of a file as a parameter and creates an input stream that can be used for reading from that file. This constructor will throw an exception of type *FileNotFoundException* if the file doesn't exist. It requires mandatory exception handling, so you have to call the constructor in a `try..catch` statement (or inside a routine that is declared to throw the exception). For example, suppose you have a file named "`data.txt`", and you want your program to read data from that file. You could do the following to create an input stream for the file:

```
FileReader data;   // (Declare the variable before the
                   //   try statement, or else the variable
                   //   is local to the try block and you won't
                   //   be able to use it later in the program.)

try {
   data = new FileReader("data.txt");  // create the stream
}
catch (FileNotFoundException e) {
   ... // do something to handle the error---maybe, end the program
}
```

The *FileNotFoundException* class is a subclass of *IOException*, so it would be acceptable to catch *IOExceptions* in the above `try...catch` statement. More generally, just about any error that can occur during input/output operations can be caught by a `catch` clause that handles *IOException*.

Once you have successfully created a *FileReader*, you can start reading data from it. But since *FileReaders* have only the primitive input methods inherited from the basic *Reader* class,

you will probably want to wrap your *FileReader* in a *Scanner*, in a *TextReader*, or in some other wrapper class. (The *TextReader* class is not a standard part of Java; it is described in Subsection 11.1.4. *Scanner* is discussed in Subsection 11.1.5.) To create a *TextReader* for reading from a file named `data.dat`, you could say:

```
TextReader data;

try {
   data = new TextReader( new FileReader("data.dat") );
}
catch (FileNotFoundException e) {
   ... // handle the exception
}
```

To use a *Scanner* to read from the file, you can construct the scanner in a similar way. However, it is more common to construct it from an object of type *File* (to be covered below):

```
Scanner in;

try {
   in = new Scanner( new File("data.dat") );
}
catch (FileNotFoundException e) {
   ... // handle the exception
}
```

Once you have a *Scanner* or *TextReader* for reading from a file, you can get data from the file using exactly the same methods that work with any *Scanner* or *TextReader*.

Working with output files is no more difficult than this. You simply create an object belonging to the class *FileWriter*. You will probably want to wrap this output stream in an object of type *PrintWriter*. For example, suppose you want to write data to a file named "`result.dat`". Since the constructor for *FileWriter* can throw an exception of type *IOException*, you should use a `try..catch` statement:

```
PrintWriter result;

try {
   result = new PrintWriter(new FileWriter("result.dat"));
}
catch (IOException e) {
   ... // handle the exception
}
```

If no file named `result.dat` exists, a new file will be created. If the file already exists, then the current contents of the file will be erased and replaced with the data that your program writes to the file. This will be done without any warning. To avoid overwriting a file that already exists, you can check whether a file of the same name already exists before trying to create the stream, as discussed later in this section. An *IOException* might occur in the *PrintWriter* constructor if, for example, you are trying to create a file on a disk that is "write-protected," meaning that it cannot be modified.

In fact, a *PrintWriter* can also be created directly from a file name given as a string ("`new PrintWriter("result.dat")`"), and you will probably find it more convenient to do that. Remember, however, that a *Scanner* for reading from a file **cannot** be created in the same way.

After you are finished using a file, it's a good idea to ***close*** the file, to tell the operating system that you are finished using it. You can close a file by calling the `close()` method of the associated stream or *Scanner*. Once a file has been closed, it is no longer possible to read data from it or write data to it, unless you open it again as a new stream. (Note that for most stream classes, the `close()` method can throw an *IOException*, which must be handled; however, *PrintWriter*, *TextReader*, and *Scanner* override this method so that it cannot throw such exceptions.) If you forget to close a file, the file will ordinarily be closed automatically when the program terminates or when the file object is garbage collected, but in the case of an output file, some of the data that has been written to the file might be lost. This can occur because data that is written to a file can be ***buffered***; that is, the data is not sent immediately to the file but is retained in main memory (in a "buffer") until a larger chunk of data is ready to be written. This is done for efficiency. The `close()` method of an output stream will cause all the data in the buffer to be sent to the file. Every output stream also has a `flush()` method that can be called to force any data in the buffer to be written to the file without closing the file.

As a complete example, here is a program that will read numbers from a file named `data.dat`, and will then write out the same numbers in reverse order to another file named `result.dat`. It is assumed that `data.dat` contains only one number on each line. Exception-handling is used to check for problems along the way. Although the application is not a particularly useful one, this program demonstrates the basics of working with files. (By the way, at the end of this program, you'll find our first useful example of a `finally` clause in a `try` statement. When the computer executes a `try` statement, the commands in its `finally` clause are guaranteed to be executed, no matter what. See Subsection 8.3.2.)

```
import java.io.*;
import java.util.ArrayList;

/**
 * Reads numbers from a file named data.dat and writes them to a file
 * named result.dat in reverse order.  The input file should contain
 * exactly one real number per line.
 */
public class ReverseFile {

   public static void main(String[] args) {

      TextReader data;     // Character input stream for reading data.
      PrintWriter result;  // Character output stream for writing data.

      ArrayList<Double> numbers;  // An ArrayList for holding the data.

      numbers = new ArrayList<Double>();

      try {  // Create the input stream.
         data = new TextReader(new FileReader("data.dat"));
      }
      catch (FileNotFoundException e) {
         System.out.println("Can't find file data.dat!");
         return;  // End the program by returning from main().
      }

      try {  // Create the output stream.
         result = new PrintWriter(new FileWriter("result.dat"));
      }
```

```
            catch (IOException e) {
               System.out.println("Can't open file result.dat!");
               System.out.println("Error: " + e);
               data.close();  // Close the input file.
               return;        // End the program.
            }

            try {

                // Read numbers from the input file, adding them to the ArrayList.

                while ( data.eof() == false ) {  // Read until end-of-file.
                   double inputNumber = data.getlnDouble();
                   numbers.add( inputNumber );
                }

                // Output the numbers in reverse order.

                for (int i = numbers.size()-1; i >= 0; i--)
                   result.println(numbers.get(i));

                System.out.println("Done!");

             }
             catch (IOException e) {
                   // Some problem reading the data from the input file.
                System.out.println("Input Error: " + e.getMessage());
             }
             finally {
                   // Finish by closing the files, whatever else may have happened.
                data.close();
                result.close();
             }

          }  // end of main()

       } // end class ReverseFile
```

A version of this program that uses a *Scanner* instead of a *TextReader* can be found in *ReverseFileWithScanner.java*. Note that the *Scanner* version does not need the final `try..catch` from the *TextReader* version, since the *Scanner* method for reading data doesn't throw an *IOException*.

11.2.2 Files and Directories

The subject of file names is actually more complicated than I've let on so far. To fully specify a file, you have to give both the name of the file and the name of the directory where that file is located. A simple file name like "data.dat" or "result.dat" is taken to refer to a file in a directory that is called the ***current directory*** (also known as the "default directory" or "working directory"). The current directory is not a permanent thing. It can be changed by the user or by a program. Files not in the current directory must be referred to by a ***path name***, which includes both the name of the file and information about the directory where it can be found.

To complicate matters even further, there are two types of path names, ***absolute path names*** and ***relative path names***. An absolute path name uniquely identifies one file among all the files available to the computer. It contains full information about which directory the

file is in and what the file's name is. A relative path name tells the computer how to locate the file starting from the current directory.

Unfortunately, the syntax for file names and path names varies somewhat from one type of computer to another. Here are some examples:

- `data.dat` — on any computer, this would be a file named "data.dat" in the current directory.
- `/home/eck/java/examples/data.dat` — This is an absolute path name in a UNIX operating system, including Linux and Mac OS X. It refers to a file named data.dat in a directory named examples, which is in turn in a directory named java, ...
- `C:\eck\java\examples\data.dat` — An absolute path name on a Windows computer.
- `Hard Drive:java:examples:data.dat` — Assuming that "Hard Drive" is the name of a disk drive, this would be an absolute path name on a computer using a classic Macintosh operating system such as Mac OS 9.
- `examples/data.dat` — a relative path name under UNIX. "examples" is the name of a directory that is contained within the current directory, and data.dat is a file in that directory. The corresponding relative path name for Windows would be `examples\data.dat`.
- `../examples/data.dat` — a relative path name in UNIX that means "go to the directory that contains the current directory, then go into a directory named examples inside that directory, and look there for a file named data.data." In general, "`..`" means "go up one directory."

It's reasonably safe to say, though, that if you stick to using simple file names only, and if the files are stored in the same directory with the program that will use them, then you will be OK. Later in this section, we'll look at a convenient way of letting the user specify a file in a GUI program, which allows you to avoid the issue of path names altogether.

It is possible for a Java program to find out the absolute path names for two important directories, the current directory and the user's home directory. The names of these directories are ***system properties***, and they can be read using the function calls:

- `System.getProperty("user.dir")` — returns the absolute path name of the current directory as a *String*.
- `System.getProperty("user.home")` — returns the absolute path name of the user's home directory as a *String*.

To avoid some of the problems caused by differences in path names between platforms, Java has the class `java.io.File`. An object belonging to this class represents a file. More precisely, an object of type *File* represents a file **name** rather than a file as such. The file to which the name refers might or might not exist. Directories are treated in the same way as files, so a *File* object can represent a directory just as easily as it can represent a file.

A *File* object has a constructor, "`new File(String)`", that creates a *File* object from a path name. The name can be a simple name, a relative path, or an absolute path. For example, `new File("data.dat")` creates a *File* object that refers to a file named data.dat, in the current directory. Another constructor, "`new File(File,String)`", has two parameters. The first is a *File* object that refers to the directory that contains the file. The second can be the name of the file or a relative path from the directory to the file.

File objects contain several useful instance methods. Assuming that `file` is a variable of type *File*, here are some of the methods that are available:

- `file.exists()` — This **boolean**-valued function returns `true` if the file named by the *File* object already exists. You can use this method if you want to avoid overwriting the contents of an existing file when you create a new *FileWriter*.
- `file.isDirectory()` — This **boolean**-valued function returns `true` if the *File* object refers to a directory. It returns `false` if it refers to a regular file or if no file with the given name exists.
- `file.delete()` — Deletes the file, if it exists. Returns a **boolean** value to indicate whether the file was successfully deleted.
- `file.list()` — If the `File` object refers to a directory, this function returns an array of type `String[]` containing the names of the files in that directory. Otherwise, it returns `null`. `file.listFiles()` is similar, except that it returns an array of *File* instead of an array of *String*

Here, for example, is a program that will list the names of all the files in a directory specified by the user. In this example, I have used a *Scanner* to read the user's input:

```
import java.io.File;
import java.util.Scanner;

/**
 * This program lists the files in a directory specified by
 * the user.  The user is asked to type in a directory name.
 * If the name entered by the user is not a directory, a
 * message is printed and the program ends.
 */
public class DirectoryList {


   public static void main(String[] args) {

      String directoryName;  // Directory name entered by the user.
      File directory;        // File object referring to the directory.
      String[] files;        // Array of file names in the directory.
      Scanner scanner;       // For reading a line of input from the user.

      scanner = new Scanner(System.in);  // scanner reads from standard input.

      System.out.print("Enter a directory name: ");
      directoryName = scanner.nextLine().trim();
      directory = new File(directoryName);

      if (directory.isDirectory() == false) {
          if (directory.exists() == false)
             System.out.println("There is no such directory!");
          else
             System.out.println("That file is not a directory.");
      }
      else {
          files = directory.list();
          System.out.println("Files in directory \"" + directory + "\":");
          for (int i = 0; i < files.length; i++)
             System.out.println("   " + files[i]);
      }

   } // end main()
```

```
} // end class DirectoryList
```

All the classes that are used for reading data from files and writing data to files have constructors that take a *File* object as a parameter. For example, if `file` is a variable of type *File*, and you want to read character data from that file, you can create a *FileReader* to do so by saying `new FileReader(file)`.

11.2.3 File Dialog Boxes

In many programs, you want the user to be able to select the file that is going to be used for input or output. If your program lets the user type in the file name, you will just have to assume that the user understands how to work with files and directories. But in a graphical user interface, the user expects to be able to select files using a ***file dialog box***, which is a window that a program can open when it wants the user to select a file for input or output. Swing includes a platform-independent technique for using file dialog boxes in the form of a class called *JFileChooser*. This class is part of the package `javax.swing`. We looked at using some basic dialog boxes in Subsection 6.8.2. File dialog boxes are similar to those, but are a little more complicated to use.

A file dialog box shows the user a list of files and sub-directories in some directory, and makes it easy for the user to specify a file in that directory. The user can also navigate easily from one directory to another. The most common constructor for *JFileChooser* has no parameter and sets the starting directory in the dialog box to be the user's home directory. There are also constructors that specify the starting directory explicitly:

```
new JFileChooser( File startDirectory )

new JFileChooser( String pathToStartDirectory )
```

Constructing a *JFileChooser* object does not make the dialog box appear on the screen. You have to call a method in the object to do that. There are two different methods that can be used because there are two types of file dialog: An ***open file dialog*** allows the user to specify an existing file to be opened for reading data into the program; a ***save file dialog*** lets the user specify a file, which might or might not already exist, to be opened for writing data from the program. File dialogs of these two types are opened using the `showOpenDialog` and `showSaveDialog` methods. These methods make the dialog box appear on the screen; the methods do not return until the user selects a file or cancels the dialog.

A file dialog box always has a ***parent***, another component which is associated with the dialog box. The parent is specified as a parameter to the `showOpenDialog` or `showSaveDialog` methods. The parent is a GUI component, and can often be specified as "`this`" in practice, since file dialogs are often used in instance methods of GUI component classes. (The parameter can also be `null`, in which case an invisible component is created to be used as the parent.) Both `showOpenDialog` and `showSaveDialog` have a return value, which will be one of the constants `JFileChooser.CANCEL_OPTION`, `JFileChooser.ERROR_OPTION`, or `JFileChooser.APPROVE_OPTION`. If the return value is `JFileChooser.APPROVE_OPTION`, then the user has selected a file. If the return value is something else, then the user did not select a file. The user might have clicked a "Cancel" button, for example. You should always check the return value, to make sure that the user has, in fact, selected a file. If that is the case, then you can find out which file was selected by calling the *JFileChooser's* `getSelectedFile()` method, which returns an object of type *File* that represents the selected file.

Putting all this together, we can look at a typical subroutine that reads data from a file that is selected using a *JFileChooser*:

```
public void readFile() {
   if (fileDialog == null)   // (fileDialog is an instance variable)
      fileDialog = new JFileChooser();
   fileDialog.setDialogTitle("Select File for Reading");
   fileDialog.setSelectedFile(null);  // No file is initially selected.
   int option = fileDialog.showOpenDialog(this);
       // (Using "this" as a parameter to showOpenDialog() assumes that the
       //  readFile() method is an instance method in a GUI component class.)
   if (option != JFileChooser.APPROVE_OPTION)
      return;  // User canceled or clicked the dialog's close box.
   File selectedFile = fileDialog.getSelectedFile();
   TextReader in;  // (or use some other wrapper class)
   try {
      FileReader stream = new FileReader(selectedFile); // (or a FileInputStream)
      in = new TextReader( stream );
   }
   catch (Exception e) {
      JOptionPane.showMessageDialog(this,
          "Sorry, but an error occurred while trying to open the file:\n" + e);
      return;
   }
   try {
      .
      .  // Read and process the data from the input stream, in.
      .
     in.close();
   }
   catch (Exception e) {
      JOptionPane.showMessageDialog(this,
          "Sorry, but an error occurred while trying to read the data:\n" + e);
   }
}
```

One fine point here is that the variable `fileDialog` is an instance variable of type *JFileChooser*. This allows the file dialog to continue to exist between calls to `readFile()`. The main effect of this is that the dialog box will keep the same selected directory from one call of `readFile()` to the next. When the dialog reappears, it will show the same directory that the user selected the previous time it appeared. This is probably what the user expects.

Note that it's common to do some configuration of a *JFileChooser* before calling `showOpenDialog` or `showSaveDialog`. For example, the instance method `setDialogTitle(String)` is used to specify a title to appear in the title bar of the window. And `setSelectedFile(File)` is used to set the file that is selected in the dialog box when it appears. This can be used to provide a default file choice for the user. In the `readFile()` method, above, `fileDialog.setSelectedFile(null)` specifies that no file is pre-selected when the dialog box appears.

Writing data to a file is similar, but it's a good idea to add a check to determine whether the output file that is selected by the user already exists. In that case, ask the user whether to replace the file. Here is a typical subroutine for writing to a user-selected file:

```
public void writeFile() {
   if (fileDialog == null)
      fileDialog = new JFileChooser();  // (fileDialog is an instance variable)
   File selectedFile = new File("output.txt"); // (default output file name)
```

```
        fileDialog.setSelectedFile(selectedFile);  // Specify a default file name.
        fileDialog.setDialogTitle("Select File for Writing");
        int option = fileDialog.showSaveDialog(this);
        if (option != JFileChooser.APPROVE_OPTION)
           return;  // User canceled or clicked the dialog's close box.
        selectedFile = fileDialog.getSelectedFile();
        if (selectedFile.exists()) {  // Ask the user whether to replace the file.
           int response = JOptionPane.showConfirmDialog( this,
                 "The file \"" + selectedFile.getName()
                     + "\" already exists.\nDo you want to replace it?",
                 "Confirm Save",
                 JOptionPane.YES_NO_OPTION,
                 JOptionPane.WARNING_MESSAGE );
           if (response != JOptionPane.YES_OPTION)
              return;  // User does not want to replace the file.
        }
        PrintWriter out;  // (or use some other wrapper class)
        try {
           FileWriter stream = new FileWriter(selectedFile); // (or FileOutputStream)
           out = new PrintWriter( stream );
        }
        catch (Exception e) {
           JOptionPane.showMessageDialog(this,
               "Sorry, but an error occurred while trying to open the file:\n" + e);
           return;
        }
        try {
           .
           .  // Write data to the output stream, out.
           .
          out.close();
          if (out.checkError())   // (need to check for errors in PrintWriter)
             throw new IOException("Error occurred while trying to write file.");
        }
        catch (Exception e) {
           JOptionPane.showMessageDialog(this,
               "Sorry, but an error occurred while trying to write the data:\n" + e);
        }
     }
```

The `readFile()` and `writeFile()` routines presented here can be used, with just a few changes, when you need to read or write a file in a GUI program. We'll look at some more complete examples of using files and file dialogs in the next section.

11.3 Programming With Files

In this section, we look at several programming examples that work with files, using the techniques that were introduced in Section 11.1 and Section 11.2.

11.3.1 Copying a File

As a first example, we look at a simple command-line program that can make a copy of a file. Copying a file is a pretty common operation, and every operating system already has a command for doing it. However, it is still instructive to look at a Java program that does the same thing. Many file operations are similar to copying a file, except that the data from the input file is processed in some way before it is written to the output file. All such operations can be done by programs with the same general form.

Since the program should be able to copy any file, we can't assume that the data in the file is in human-readable form. So, we have to use *InputStream* and *OutputStream* to operate on the file rather than *Reader* and *Writer*. The program simply copies all the data from the *InputStream* to the *OutputStream*, one byte at a time. If `source` is the variable that refers to the *InputStream*, then the function `source.read()` can be used to read one byte. This function returns the value -1 when all the bytes in the input file have been read. Similarly, if `copy` refers to the *OutputStream*, then `copy.write(b)` writes one byte to the output file. So, the heart of the program is a simple `while` loop. As usual, the I/O operations can throw exceptions, so this must be done in a `try..catch` statement:

```
while(true) {
   int data = source.read();
   if (data < 0)
      break;
   copy.write(data);
}
```

The file-copy command in an operating system such as UNIX uses command line arguments to specify the names of the files. For example, the user might say "`copy original.dat backup.dat`" to copy an existing file, `original.dat`, to a file named `backup.dat`. Command-line arguments can also be used in Java programs. The command line arguments are stored in the array of strings, `args`, which is a parameter to the `main()` routine. The program can retrieve the command-line arguments from this array. (See Subsection 7.2.3.) For example, if the program is named `CopyFile` and if the user runs the program with the command "`java CopyFile work.dat oldwork.dat`", then in the program, `args[0]` will be the string `"work.dat"` and `args[1]` will be the string `"oldwork.dat"`. The value of `args.length` tells the program how many command-line arguments were specified by the user.

My `CopyFile` program gets the names of the files from the command-line arguments. It prints an error message and exits if the file names are not specified. To add a little interest, there are two ways to use the program. The command line can simply specify the two file names. In that case, if the output file already exists, the program will print an error message and end. This is to make sure that the user won't accidently overwrite an important file. However, if the command line has three arguments, then the first argument must be "`-f`" while the second and third arguments are file names. The `-f` is a ***command-line option***, which is meant to modify the behavior of the program. The program interprets the `-f` to mean that it's OK to overwrite an existing program. (The "f" stands for "force," since it forces the file to be copied in spite of what would otherwise have been considered an error.) You can see in the source code how the command line arguments are interpreted by the program:

```
import java.io.*;

/**
 *  Makes a copy of a file.  The original file and the name of the
```

```
 *  copy must be given as command-line arguments.  In addition, the
 *  first command-line argument can be "-f"; if present, the program
 *  will overwrite an existing file; if not, the program will report
 *  an error and end if the output file already exists.  The number
 *  of bytes that are copied is reported.
 */
public class CopyFile {

   public static void main(String[] args) {

      String sourceName;   // Name of the source file,
                           //    as specified on the command line.
      String copyName;     // Name of the copy,
                           //    as specified on the command line.
      InputStream source;  // Stream for reading from the source file.
      OutputStream copy;   // Stream for writing the copy.
      boolean force;  // This is set to true if the "-f" option
                      //    is specified on the command line.
      int byteCount;  // Number of bytes copied from the source file.

      /* Get file names from the command line and check for the
         presence of the -f option.  If the command line is not one
         of the two possible legal forms, print an error message and
         end this program. */

      if (args.length == 3 && args[0].equalsIgnoreCase("-f")) {
         sourceName = args[1];
         copyName = args[2];
         force = true;
      }
      else if (args.length == 2) {
         sourceName = args[0];
         copyName = args[1];
         force = false;
      }
      else {
         System.out.println(
                 "Usage:  java CopyFile <source-file> <copy-name>");
         System.out.println(
                 "    or  java CopyFile -f <source-file> <copy-name>");
         return;
      }

      /* Create the input stream.  If an error occurs, end the program. */

      try {
         source = new FileInputStream(sourceName);
      }
      catch (FileNotFoundException e) {
         System.out.println("Can't find file \"" + sourceName + "\".");
         return;
      }

      /* If the output file already exists and the -f option was not
         specified, print an error message and end the program. */

      File file = new File(copyName);
```

```
            if (file.exists() && force == false) {
                System.out.println(
                     "Output file exists.  Use the -f option to replace it.");
                return;
            }

            /* Create the output stream.  If an error occurs, end the program. */

            try {
               copy = new FileOutputStream(copyName);
            }
            catch (IOException e) {
               System.out.println("Can't open output file \"" + copyName + "\".");
               return;
            }

            /* Copy one byte at a time from the input stream to the output
               stream, ending when the read() method returns -1 (which is
               the signal that the end of the stream has been reached).  If any
               error occurs, print an error message.  Also print a message if
               the file has been copied successfully.  */

            byteCount = 0;

            try {
               while (true) {
                  int data = source.read();
                  if (data < 0)
                     break;
                  copy.write(data);
                  byteCount++;
               }
               source.close();
               copy.close();
               System.out.println("Successfully copied " + byteCount + " bytes.");
            }
            catch (Exception e) {
               System.out.println("Error occurred while copying.  "
                                        + byteCount + " bytes copied.");
               System.out.println("Error: " + e);
            }

         }  // end main()

      } // end class CopyFile
```

It is not terribly efficient to copy one byte at a time. Efficiency could be improved by using alternative versions of the `read()` and `write()` methods that read and write multiple bytes (see the API for details). Alternatively, the input and output streams could be wrapped in objects of type *BufferedInputStream* and *BufferedOutputStream* which automatically read from and write data to files in larger blocks, which is more efficient than reading and writing individual bytes.

11.3.2 Persistent Data

Once a program ends, any data that was stored in variables and objects in the program is gone. In many cases, it would be useful to have some of that data stick around so that it will be available when the program is run again. The problem is, how to make the data ***persistent*** between runs of the program? The answer, of course, is to store the data in a file (or, for some applications, in a database—but the data in a database is itself stored in files).

Consider a "phone book" program that allows the user to keep track of a list of names and associated phone numbers. The program would make no sense at all if the user had to create the whole list from scratch each time the program is run. It would make more sense to think of the phone book as a persistent collection of data, and to think of the program as an interface to that collection of data. The program would allow the user to look up names in the phone book and to add new entries. Any changes that are made should be preserved after the program ends.

The sample program *PhoneDirectoryFileDemo.java* is a very simple implementation of this idea. It is meant only as an example of file use; the phone book that it implements is a "toy" version that is not meant to be taken seriously. This program stores the phone book data in a file named "`.phone_book_demo`" in the user's home directory. To find the user's home directory, it uses the `System.getProperty()` method that was mentioned in Subsection 11.2.2. When the program starts, it checks whether the file already exists. If it does, it should contain the user's phone book, which was saved in a previous run of the program, so the data from the file is read and entered into a *TreeMap* named `phoneBook` that represents the phone book while the program is running. (See Subsection 10.3.1.) In order to store the phone book in a file, some decision must be made about how the data in the phone book will be represented. For this example, I chose a simple representation in which each line of the file contains one entry consisting of a name and the associated phone number. A percent sign (`'%'`) separates the name from the number. The following code at the beginning of the program will read the phone book data file, if it exists and has the correct format:

```
File userHomeDirectory = new File( System.getProperty("user.home") );
File dataFile = new File( userHomeDirectory, ".phone_book_data" );

if ( ! dataFile.exists() ) {
   System.out.println("No phone book data file found.");
   System.out.println("A new one will be created.");
   System.out.println("File name:  " + dataFile.getAbsolutePath());
}
else {
   System.out.println("Reading phone book data...");
   try {
      Scanner scanner = new Scanner( dataFile );
      while (scanner.hasNextLine()) {
             // Read one line from the file, containing one name/number pair.
         String phoneEntry = scanner.nextLine();
         int separatorPosition = phoneEntry.indexOf('%');
         if (separatorPosition == -1)
            throw new IOException("File is not a phonebook data file.");
         name = phoneEntry.substring(0, separatorPosition);
         number = phoneEntry.substring(separatorPosition+1);
         phoneBook.put(name,number);
      }
```

```
        }
        catch (IOException e) {
           System.out.println("Error in phone book data file.");
           System.out.println("File name:  " + dataFile.getAbsolutePath());
           System.out.println("This program cannot continue.");
           System.exit(1);
        }
    }
```

The program then lets the user do various things with the phone book, including making modifications. Any changes that are made are made only to the *TreeMap* that holds the data. When the program ends, the phone book data is written to the file (if any changes have been made while the program was running), using the following code:

```
    if (changed) {
       System.out.println("Saving phone directory changes to file " +
             dataFile.getAbsolutePath() + " ...");
       PrintWriter out;
       try {
          out = new PrintWriter( new FileWriter(dataFile) );
       }
       catch (IOException e) {
          System.out.println("ERROR: Can't open data file for output.");
          return;
       }
       for ( Map.Entry<String,String> entry : phoneBook.entrySet() )
          out.println(entry.getKey() + "%" + entry.getValue() );
       out.close();
       if (out.checkError())
          System.out.println("ERROR: Some error occurred while writing data file.");
       else
          System.out.println("Done.");
    }
```

The net effect of this is that all the data, including the changes, will be there the next time the program is run. I've shown you all the file-handling code from the program. If you would like to see the rest of the program, see the source code file, *PhoneDirectoryFileDemo.java*.

11.3.3 Files in GUI Programs

The previous examples in this section use a command-line interface, but graphical user interface programs can also manipulate files. Programs typically have an "Open" command that reads the data from a file and displays it in a window and a "Save" command that writes the data from the window into a file. We can illustrate this in Java with a simple text editor program, *TrivialEdit.java*. The window for this program uses a *JTextArea* component to display some text that the user can edit. It also has a menu bar, with a "File" menu that includes "Open" and "Save" commands. These commands are implemented using the techniques for reading and writing files that were covered in Section 11.2.

When the user selects the Open command from the File menu in the `TrivialEdit` program, the program pops up a file dialog box where the user specifies the file. It is assumed that the file is a text file. A limit of 10000 characters is put on the size of the file, since a *JTextArea* is not meant for editing large amounts of text. The program reads the text contained in the

specified file, and sets that text to be the content of the *JTextArea*. In this case, I decided to use a *BufferedReader* to read the file line-by-line. The program also sets the title bar of the window to show the name of the file that was opened. All this is done in the following method, which is just a variation of the `readFile()` method presented in Section 11.2:

```
/**
 * Carry out the Open command by letting the user specify a file to be opened
 * and reading up to 10000 characters from that file.  If the file is read
 * successfully and is not too long, then the text from the file replaces the
 * text in the JTextArea.
 */
public void doOpen() {
   if (fileDialog == null)
      fileDialog = new JFileChooser();
   fileDialog.setDialogTitle("Select File to be Opened");
   fileDialog.setSelectedFile(null);  // No file is initially selected.
   int option = fileDialog.showOpenDialog(this);
   if (option != JFileChooser.APPROVE_OPTION)
      return;  // User canceled or clicked the dialog's close box.
   File selectedFile = fileDialog.getSelectedFile();
   BufferedReader in;
   try {
      FileReader stream = new FileReader(selectedFile);
      in = new BufferedReader( stream );
   }
   catch (Exception e) {
      JOptionPane.showMessageDialog(this,
            "Sorry, but an error occurred while trying to open the file:\n" + e);
      return;
   }
   try {
      StringBuffer input = new StringBuffer();
         while (true) {
            String lineFromFile = in.readLine();
            if (lineFromFile == null)
               break;  // End-of-file has been reached.
            input.append(lineFromFile);
            input.append('\n');
            if (input.length() > 10000)
               throw new IOException("Input file is too large for this program.");
         }
      in.close();
      text.setText(input);
      editFile = selectedFile;
      setTitle("TrivialEdit: " + editFile.getName());
   }
   catch (Exception e) {
      JOptionPane.showMessageDialog(this,
            "Sorry, but an error occurred while trying to read the data:\n" + e);
   }
}
```

In this program, the instance variable `editFile` is used to keep track of the file that is currently being edited, if any, and the `setTitle()` method (from class *JFrame*) is used to set the title of the window to show the name of the file.

Similarly, the response to the Save command is a minor variation on the `writeFile()` method from Section 11.2. I will not repeat it here. If you would like to see the entire program, you will find the source code in the file *TrivialEdit.java*.

11.3.4 Storing Objects in Files

Whenever data is stored in files, some definite format must be adopted for representing the data. As long as the output routine that writes the data and the input routine that reads the data use the same format, the files will be usable. However, as usual, correctness is not the end of the story. The representation that is used for data in files should also be robust. (See Section 8.1.) To see what this means, we will look at several different ways of representing the same data. This example builds on the example *SimplePaint2.java* from Subsection 7.3.4. In that program, the user could use the mouse to draw simple sketches. Now, we will add file input/output capabilities to that program. This will allow the user to save a sketch to a file and later read the sketch back from the file into the program so that the user can continue to work on the sketch. The basic requirement is that all relevant data about the sketch must be saved in the file, so that the sketch can be exactly restored when the file is read by the program.

The new version of the program can be found in the source code file *SimplePaintWithFiles.java*. A "File" menu has been added to the new version. It contains two sets of Save/Open commands, one for saving and reloading sketch data in text form and one for data in binary form. We will consider both possibilities here, in some detail.

The data for a sketch consists of the background color of the picture and a list of the curves that were drawn by the user. A curve consists of a list of *Points*. (*Point* is a standard class in package `java.awt`; a *Point* `pt` has instance variables `pt.x` and `pt.y` of type **int** that represent the coordinates of a point on the xy-plane.) Each curve can be a different color. Furthermore, a curve can be "symmetric," which means that in addition to the curve itself, the horizontal and vertical reflections of the curve are also drawn. The data for each curve is stored in an object of type *CurveData*, which is defined in the program as:

```
/**
 * An object of type CurveData represents the data required to redraw one
 * of the curves that have been sketched by the user.
 */
private static class CurveData implements Serializable {
   Color color;  // The color of the curve.
   boolean symmetric;  // Are horizontal and vertical reflections also drawn?
   ArrayList<Point> points;  // The points on the curve.
}
```

Note that this class has been declared to "`implement Serializable`". This allows objects of type *CurveData* to be written in binary form to an *ObjectOutputStream*. See Subsection 11.1.6.

Let's think about how the data for a sketch could be saved to an *ObjectOuputStream*. The sketch is displayed on the screen in an object of type *SimplePaintPanel*, which is a subclass of *JPanel*. All the data needed for the sketch is stored in instance variables of that object. One possibility would be to simply write the entire *SimplePaintPanel* component as a single object to the stream. This could be done in a method in the *SimplePaintPanel* class with the statement

```
outputStream.writeObject(this);
```

where `outputStream` is the *ObjectOutputStream* and "`this`" refers to the *SimplePaintPanel* itself. This statement saves the entire current state of the panel. To read the data back into the program, you would create an *ObjectInputStream* for reading the object from the file, and you would retrieve the object from the file with the statement

```
SimplePaintPanel newPanel = (SimplePaintPanel)in.readObject();
```

where `in` is the `ObjectInputStream`. Note that the type-cast is necessary because the method `in.readObject()` returns a value of type *Object*. (To get the saved sketch to appear on the screen, the `newPanel` must replace the current content pane in the program's window; furthermore, the menu bar of the window must be replaced, because the menus are associated with a particular *SimplePaintPanel* object.)

It might look tempting to be able to save data and restore it with a single command, but in this case, it's not a good idea. The main problem with doing things this way is that **the serialized form of objects that represent Swing components can change** from one version of Java to the next. This means that data files that contain serialized components such as a *SimplePaintPanel* might become unusable in the future, and the data that they contain will be effectively lost. This is an important consideration for any serious application.

Taking this into consideration, my program uses a different format when it creates a binary file. The data written to the file consists of (1) the background color of the sketch, (2) the number of curves in the sketch, and (3) all the *CurveData* objects that describe the individual curves. The method that saves the data is similar to the `writeFile()` method from Subsection 11.2.3. Here is the complete `doSaveAsBinary()` method from `SimplePaintWithFiles`, with the changes from the generic `readFile()` method shown in italic:

```
/**
 * Save the user's sketch to a file in binary form as serialized
 * objects, using an ObjectOutputStream.  Files created by this method
 * can be read back into the program using the doOpenAsBinary() method.
 */
private void doSaveAsBinary() {
   if (fileDialog == null)
      fileDialog = new JFileChooser();
   File selectedFile;  //Initially selected file name in the dialog.
   if (editFile == null)
      selectedFile = new File("sketchData.binary");
   else
      selectedFile = new File(editFile.getName());
   fileDialog.setSelectedFile(selectedFile);
   fileDialog.setDialogTitle("Select File to be Saved");
   int option = fileDialog.showSaveDialog(this);
   if (option != JFileChooser.APPROVE_OPTION)
      return;  // User canceled or clicked the dialog's close box.
   selectedFile = fileDialog.getSelectedFile();
   if (selectedFile.exists()) {  // Ask the user whether to replace the file.
      int response = JOptionPane.showConfirmDialog( this,
            "The file \"" + selectedFile.getName()
            + "\" already exists.\nDo you want to replace it?",
            "Confirm Save",
            JOptionPane.YES_NO_OPTION,
            JOptionPane.WARNING_MESSAGE );
      if (response != JOptionPane.YES_OPTION)
```

```
            return;  // User does not want to replace the file.
        }
        ObjectOutputStream out;
        try {
            FileOutputStream stream = new FileOutputStream(selectedFile);
            out = new ObjectOutputStream( stream );
        }
        catch (Exception e) {
            JOptionPane.showMessageDialog(this,
               "Sorry, but an error occurred while trying to open the file:\n" + e);
            return;
        }
        try {
            out.writeObject(getBackground());
            out.writeInt(curves.size());
            for ( CurveData curve : curves )
               out.writeObject(curve);
            out.close();
            editFile = selectedFile;
            setTitle("SimplePaint: " + editFile.getName());
        }
        catch (Exception e) {
            JOptionPane.showMessageDialog(this,
               "Sorry, but an error occurred while trying to write the text:\n" + e);
        }
    }
```

The heart of this method consists of the following lines, which do the actual writing of the data to the file:

```
out.writeObject(getBackground()); // Writes the panel's background color.
out.writeInt(curves.size());      // Writes the number of curves.
for ( CurveData curve : curves )  // For each curve...
   out.writeObject(curve);        //   write the corresponding CurveData object.
```

The last line depends on the fact that the *CurveData* class implements the *Serializable* interface.

The `doOpenAsBinary()` method, which is responsible for reading sketch data back into the program from an *ObjectInputStream*, has to read exactly the same data that was written, in the same order, and use that data to build the data structures that will represent the sketch while the program is running. Once the data structures have been successfully built, they replace the data structures that describe the previous contents of the panel. This is done as follows:

```
/* Read data from the file into local variables */

Color newBackgroundColor = (Color)in.readObject();
int curveCount = in.readInt();
ArrayList<CurveData> newCurves = new ArrayList<CurveData>();
for (int i = 0; i < curveCount; i++)
   newCurves.add( (CurveData)in.readObject() );
in.close();

/* Copy the data that was read into the instance variables that
   describe the sketch that is displayed by the program.*/

curves = newCurves;
setBackground(newBackgroundColor);
```

```
repaint();
```

This is only a little harder than saving the entire *SimplePaintPanel* component to the file in one step, and it is more robust since the serialized form of the objects that are saved to file is unlikely to change in the future. But it still suffers from the general fragility of binary data.

* * *

An alternative to using object streams is to save the data in human-readable, character form. The basic idea is the same: All the data necessary to reconstitute a sketch must be saved to the output file in some definite format. The method that reads the file must follow exactly the same format as it reads the data, and it must use the data to rebuild the data structures that represent the sketch while the program is running.

When writing character data, we can't write out entire objects in one step. All the data has to be expressed, ultimately, in terms of simple data values such as strings and primitive type values. A color, for example, can be expressed in terms of three integers giving the red, green, and blue components of the color. The first (not very good) idea that comes to mind might be to just dump all the necessary data, in some definite order, into the file. Suppose that `out` is a *PrintWriter* that is used to write to the file. We could then say:

```
Color bgColor = getBackground();    // Write the background color to the file.
out.println( bgColor.getRed() );
out.println( bgColor.getGreen() );
out.println( bgColor.getBlue() );

out.println( curves.size() );       // Write the number of curves.

for ( CurveData curve : curves ) {  // For each curve, write...
   out.println( curve.color.getRed() );      // the color of the curve
   out.println( curve.color.getGreen() );
   out.println( curve.color.getBlue() );
   out.println( curve.symmetric ? 0 : 1 );   // the curve's symmetry property
   out.println( curve.points.size() );       // the number of points on curve
   for ( Point pt : curve.points ) {         // the coordinates of each point
      out.println( pt.x );
      out.println( pt.y );
   }
}
```

This works in the sense that the file-reading method can read the data and rebuild the data structures. Suppose that the input method uses a *Scanner* named `scanner` to read the data file. Then it could say:

```
Color newBackgroundColor;                // Read the background Color.
int red = scanner.nextInt();
int green = scanner.nextInt();
int blue = scanner.nextInt();
newBackgroundColor = new Color(red,green,blue);

ArrayList<CurveData> newCurves = new ArrayList<CurveData>();

int curveCount = scanner.nextInt();      // The number of curves to be read.
for (int i = 0; i < curveCount; i++) {
   CurveData curve = new CurveData();
   int r = scanner.nextInt();            // Read the curve's color.
   int g = scanner.nextInt();
```

```
            int b = scanner.nextInt();
            curve.color = new Color(r,g,b);
            int symmetryCode = scanner.nextInt(); // Read the curve's symmetry property.
            curve.symmetric = (symmetryCode == 1);
            curveData.points = new ArrayList<Point>();
            int pointCount = scanner.nextInt();  // The number of points on this curve.
            for (int j = 0; j < pointCount; j++) {
               int x = scanner.nextInt();        // Read the coordinates of the point.
               int y = scanner.nextInt();
               curveData.points.add(new Point(x,y));
            }
            newCurves.add(curve);
         }

         curves = newCurves;                     // Install the new data structures.
         setBackground(newBackgroundColor);
```

Note how every piece of data that was written by the output method is read, in the same order, by the input method. While this does work, the data file is just a long string of numbers. It doesn't make much more sense to a human reader than a binary-format file would. Furthermore, it is still fragile in the sense that any small change made to the data representation in the program, such as adding a new property to curves, will render the data file useless (unless you happen to remember exactly which version of the program created the file).

So, I decided to use a more complex, more meaningful data format for the text files created by my program. Instead of just writing numbers, I add **words** to say what the numbers mean. Here is a short but complete data file for the program; just by looking at it, you can probably tell what is going on:

```
SimplePaintWithFiles 1.0
background 110 110 180

startcurve
  color 255 255 255
  symmetry true
  coords 10 10
  coords 200 250
  coords 300 10
endcurve

startcurve
  color 0 255 255
  symmetry false
  coords 10 400
  coords 590 400
endcurve
```

The first line of the file identifies the program that created the data file; when the user selects a file to be opened, the program can check the first word in the file as a simple test to make sure the file is of the correct type. The first line also contains a version number, 1.0. If the file format changes in a later version of the program, a higher version number would be used; if the program sees a version number of 1.2 in a file, but the program only understands version 1.0, the program can explain to the user that a newer version of the program is needed to read the data file.

The second line of the file specifies the background color of the picture. The three integers specify the red, green, and blue components of the color. The word "background" at the beginning of the line makes the meaning clear. The remainder of the file consists of data for the curves that appear in the picture. The data for each curve is clearly marked with "startcurve" and "endcurve." The data consists of the color and symmetry properties of the curve and the xy-coordinates of each point on the curve. Again, the meaning is clear. Files in this format can easily be created or edited by hand. In fact, the data file shown above was actually created in a text editor rather than by the program. Furthermore, it's easy to extend the format to allow for additional options. Future versions of the program could add a "thickness" property to the curves to make it possible to have curves that are more than one pixel wide. Shapes such as rectangles and ovals could easily be added.

Outputting data in this format is easy. Suppose that `out` is a *PrintWriter* that is being used to write the sketch data to a file. Then the output can be done with:

```
out.println("SimplePaintWithFiles 1.0"); // Version number.
Color bgColor = getBackground();
out.println( "background " + bgColor.getRed() + " " +
      bgColor.getGreen() + " " + bgColor.getBlue() );
for ( CurveData curve : curves ) {
   out.println();
   out.println("startcurve");
   out.println("  color " + curve.color.getRed() + " " +
         curve.color.getGreen() + " " + curve.color.getBlue() );
   out.println( "  symmetry " + curve.symmetric );
   for ( Point pt : curve.points )
      out.println( "  coords " + pt.x + " " + pt.y );
   out.println("endcurve");
}
```

Reading the data is somewhat harder, since the input routine has to deal with all the extra words in the data. In my input routine, I decided to allow some variation in the order in which the data occurs in the file. For example, the background color can be specified at the end of the file, instead of at the beginning. It can even be left out altogether, in which case white will be used as the default background color. This is possible because each item of data is labeled with a word that describes its meaning; the labels can be used to drive the processing of the input. Here is the complete method from *SimplePaintWithFiles.java* that reads data files in text format. It uses a *Scanner* to read items from the file:

```
private void doOpenAsText() {

   if (fileDialog == null)
      fileDialog = new JFileChooser();
   fileDialog.setDialogTitle("Select File to be Opened");
   fileDialog.setSelectedFile(null);  // No file is initially selected.
   int option = fileDialog.showOpenDialog(this);
   if (option != JFileChooser.APPROVE_OPTION)
      return;  // User canceled or clicked the dialog's close box.
   File selectedFile = fileDialog.getSelectedFile();

   Scanner scanner;  // For reading from the data file.
   try {
      Reader stream = new BufferedReader(new FileReader(selectedFile));
      scanner = new Scanner( stream );
```

```
}
catch (Exception e) {
   JOptionPane.showMessageDialog(this,
         "Sorry, but an error occurred while trying to open the file:\n" + e);
   return;
}

try {  // Read the contents of the file.
   String programName = scanner.next();
   if ( ! programName.equals("SimplePaintWithFiles") )
      throw new IOException("File is not a SimplePaintWithFiles data file.");
   double version = scanner.nextDouble();
   if (version > 1.0)
      throw new IOException("File requires newer version of this program.");
   Color newBackgroundColor = Color.WHITE;
   ArrayList<CurveData> newCurves = new ArrayList<CurveData>();
   while (scanner.hasNext()) {
      String itemName = scanner.next();
      if (itemName.equalsIgnoreCase("background")) {
         int red = scanner.nextInt();
         int green = scanner.nextInt();
         int blue = scanner.nextInt();
         newBackgroundColor = new Color(red,green,blue);
      }
      else if (itemName.equalsIgnoreCase("startcurve")) {
         CurveData curve = new CurveData();
         curve.color = Color.BLACK;
         curve.symmetric = false;
         curve.points = new ArrayList<Point>();
         itemName = scanner.next();
         while ( ! itemName.equalsIgnoreCase("endcurve") ) {
            if (itemName.equalsIgnoreCase("color")) {
               int r = scanner.nextInt();
               int g = scanner.nextInt();
               int b = scanner.nextInt();
               curve.color = new Color(r,g,b);
            }
            else if (itemName.equalsIgnoreCase("symmetry")) {
               curve.symmetric = scanner.nextBoolean();
            }
            else if (itemName.equalsIgnoreCase("coords")) {
               int x = scanner.nextInt();
               int y = scanner.nextInt();
               curve.points.add( new Point(x,y) );
            }
            else {
               throw new Exception("Unknown term in input.");
            }
            itemName = scanner.next();
         }
         newCurves.add(curve);
      }
      else {
         throw new Exception("Unknown term in input.");
```

```
            }
         }

         scanner.close();
         setBackground(newBackgroundColor);  // Install the new picture data.
         curves = newCurves;
         repaint();
         editFile = selectedFile;
         setTitle("SimplePaint: " + editFile.getName());
      }
      catch (Exception e) {
         JOptionPane.showMessageDialog(this,
               "Sorry, but an error occurred while trying to read the data:\n" + e);
      }
   }
```

The main reason for this long discussion of file formats has been to get you to think about the problem of representing complex data in a form suitable for storing the data in a file. The same problem arises when data must be transmitted over a network. There is no one correct solution to the problem, but some solutions are certainly better than others. In Section 11.5, we will look at one solution to the data representation problem that has become increasingly common.

* * *

In addition to being able to save sketch data in both text form and binary form, `SimplePaintWithFiles` can also save the picture itself as an image file that could be, for example, printed or put on a web page. This is a preview of image-handling techniques that will be covered in Chapter 13.

11.4 Networking

As far as a program is concerned, a network is just another possible source of input data, and another place where data can be output. That does oversimplify things, because networks are not as easy to work with as files are. But in Java, you can do network communication using input streams and output streams, just as you can use such streams to communicate with the user or to work with files. Nevertheless, opening a network connection between two computers is a bit tricky, since there are two computers involved and they have to somehow agree to open a connection. And when each computer can send data to the other, synchronizing communication can be a problem. But the fundamentals are the same as for other forms of I/O.

One of the standard Java packages is called `java.net`. This package includes several classes that can be used for networking. Two different styles of network I/O are supported. One of these, which is fairly high-level, is based on the World-Wide Web, and provides the sort of network communication capability that is used by a Web browser when it downloads pages for you to view. The main classes for this style of networking are `java.net.URL` and `java.net.URLConnection`. An object of type *URL* is an abstract representation of a ***Universal Resource Locator***, which is an address for an HTML document or other resource on the Web. A *URLConnection* represents a network connection to such a resource.

The second style of I/O, which is more general and much more important, views the network at a lower level. It is based on the idea of a ***socket***. A socket is used by a program to establish a connection with another program on a network. Communication over a network involves two

sockets, one on each of the computers involved in the communication. Java uses a class called `java.net.Socket` to represent sockets that are used for network communication. The term "socket" presumably comes from an image of physically plugging a wire into a computer to establish a connection to a network, but it is important to understand that a socket, as the term is used here, is simply an object belonging to the class *Socket*. In particular, a program can have several sockets at the same time, each connecting it to another program running on some other computer on the network. All these connections use the same physical network connection.

This section gives a brief introduction to these basic networking classes, and shows how they relate to input and output streams.

11.4.1 URLs and URLConnections

The *URL* class is used to represent resources on the World-Wide Web. Every resource has an address, which identifies it uniquely and contains enough information for a Web browser to find the resource on the network and retrieve it. The address is called a "url" or "universal resource locator."

An object belonging to the *URL* class represents such an address. Once you have a *URL* object, you can use it to open a *URLConnection* to the resource at that address. A url is ordinarily specified as a string, such as "`http://math.hws.edu/eck/index.html`". There are also ***relative url's***. A relative url specifies the location of a resource relative to the location of another url, which is called the ***base*** or ***context*** for the relative url. For example, if the context is given by the url `http://math.hws.edu/eck/`, then the incomplete, relative url "`index.html`" would really refer to `http://math.hws.edu/eck/index.html`.

An object of the class *URL* is not simply a string, but it can be constructed from a string representation of a url. A *URL* object can also be constructed from another *URL* object, representing a context, and a string that specifies a url relative to that context. These constructors have prototypes

```
public URL(String urlName) throws MalformedURLException
```

and

```
public URL(URL context, String relativeName) throws MalformedURLException
```

Note that these constructors will throw an exception of type *MalformedURLException* if the specified strings don't represent legal url's. The *MalformedURLException* class is a subclass of *IOException*, and it requires mandatory exception handling. That is, you must call the constructor inside a `try..catch` statement that handles the exception or in a subroutine that is declared to throw the exception.

The second constructor is especially convenient when writing applets. In an applet, two methods are available that provide useful URL contexts. The method `getDocumentBase()`, defined in the *Applet* and *JApplet* classes, returns an object of type *URL*. This *URL* represents the location from which the HTML page that contains the applet was downloaded. This allows the applet to go back and retrieve other files that are stored in the same location as that document. For example,

```
URL url = new URL(getDocumentBase(), "data.txt");
```

constructs a *URL* that refers to a file named `data.txt` on the same computer and in the same directory as the source file for the web page on which the applet is running. Another method,

`getCodeBase()`, returns a *URL* that gives the location of the applet class file (which is not necessarily the same as the location of the document).

Once you have a valid *URL* object, you can call its `openConnection()` method to set up a connection. This method returns a *URLConnection*. The *URLConnection* object can, in turn, be used to create an *InputStream* for reading data from the resource represented by the URL. This is done by calling its `getInputStream()` method. For example:

```
URL url = new URL(urlAddressString);
URLConnection connection = url.openConnection();
InputStream in = connection.getInputStream();
```

The `openConnection()` and `getInputStream()` methods can both throw exceptions of type *IOException*. Once the *InputStream* has been created, you can read from it in the usual way, including wrapping it in another input stream type, such as *BufferedReader*, or using a *Scanner*. Reading from the stream can, of course, generate exceptions.

One of the other useful instance methods in the *URLConnection* class is `getContentType()`, which returns a *String* that describes the type of information available from the URL. The return value can be `null` if the type of information is not yet known or if it is not possible to determine the type. The type might not be available until after the input stream has been created, so you should generally call `getContentType()` after `getInputStream()`. The string returned by `getContentType()` is in a format called a ***mime type***. Mime types include "text/plain", "text/html", "image/jpeg", "image/gif", and many others. All mime types contain two parts: a general type, such as "text" or "image", and a more specific type within that general category, such as "html" or "gif". If you are only interested in text data, for example, you can check whether the string returned by `getContentType()` starts with "text". (Mime types were first introduced to describe the content of email messages. The name stands for "Multipurpose Internet Mail Extensions." They are now used almost universally to specify the type of information in a file or other resource.)

Let's look at a short example that uses all this to read the data from a URL. This subroutine opens a connection to a specified URL, checks that the type of data at the URL is text, and then copies the text onto the screen. Many of the operations in this subroutine can throw exceptions. They are handled by declaring that the subroutine "`throws IOException`" and leaving it up to the main program to decide what to do when an error occurs.

```
static void readTextFromURL( String urlString ) throws IOException {

   /* Open a connection to the URL, and get an input stream
      for reading data from the URL. */

   URL url = new URL(urlString);
   URLConnection connection = url.openConnection();
   InputStream urlData = connection.getInputStream();

   /* Check that the content is some type of text. */

   String contentType = connection.getContentType();
   if (contentType == null || contentType.startsWith("text") == false)
      throw new IOException("URL does not seem to refer to a text file.");

   /* Copy lines of text from the input stream to the screen, until
      end-of-file is encountered (or an error occurs). */

   BufferedReader in;  // For reading from the connection's input stream.
   in = new BufferedReader( new InputStreamReader(urlData) );
```

```
            while (true) {
               String line = in.readLine();
               if (line == null)
                  break;
               System.out.println(line);
            }

         } // end readTextFromURL()
```

A complete program that uses this subroutine can be found in the file *ReadURL.java*. When using the program, note that you have to specify a complete url, including the "`http://`" at the beginning. There is also an applet version of the program, which you can find in the on-line version of this section.

11.4.2 TCP/IP and Client/Server

Communication over the Internet is based on a pair of protocols called the ***Transmission Control Protocol*** and the ***Internet Protocol***, which are collectively referred to as ***TCP/IP***. (In fact, there is a more basic communication protocol called UDP that can be used instead of TCP in certain applications. UDP is supported in Java, but for this discussion, I'll stick to the full TCP/IP, which provides reliable two-way communication between networked computers.)

For two programs to communicate using TCP/IP, each program must create a socket, as discussed earlier in this section, and those sockets must be connected. Once such a connection is made, communication takes place using input streams and output streams. Each program has its own input stream and its own output stream. Data written by one program to its output stream is transmitted to the other computer. There, it enters the input stream of the program at the other end of the network connection. When that program reads data from its input stream, it is receiving the data that was transmitted to it over the network.

The hard part, then, is making a network connection in the first place. Two sockets are involved. To get things started, one program must create a socket that will wait passively until a connection request comes in from another socket. The waiting socket is said to be ***listening*** for a connection. On the other side of the connection-to-be, another program creates a socket that sends out a connection request to the listening socket. When the listening socket receives the connection request, it responds, and the connection is established. Once that is done, each program can obtain an input stream and an output stream for sending data over the connection. Communication takes place through these streams until one program or the other ***closes*** the connection.

A program that creates a listening socket is sometimes said to be a ***server***, and the socket is called a ***server socket***. A program that connects to a server is called a ***client***, and the socket that it uses to make a connection is called a ***client socket***. The idea is that the server is out there somewhere on the network, waiting for a connection request from some client. The server can be thought of as offering some kind of service, and the client gets access to that service by connecting to the server. This is called the ***client/server model*** of network communication. In many actual applications, a server program can provide connections to several clients at the same time. When a client connects to a server's listening socket, that socket does not stop listening. Instead, it continues listening for additional client connections at the same time that the first client is being serviced. To do this, it is necessary to use threads. We'll look at how it works in the next chapter.

The *URL* class that was discussed at the beginning of this section uses a client socket behind the scenes to do any necessary network communication. On the other side of that connection is a server program that accepts a connection request from the *URL* object, reads a request from that object for some particular file on the server computer, and responds by transmitting the contents of that file over the network back to the *URL* object. After transmitting the data, the server closes the connection.

* * *

A client program has to have some way to specify which computer, among all those on the network, it wants to communicate with. Every computer on the Internet has an ***IP address*** which identifies it uniquely among all the computers on the net. Many computers can also be referred to by ***domain names*** such as math.hws.edu or www.whitehouse.gov. (See Section 1.7.) Traditional (or ***IPv4***) IP addresses are 32-bit integers. They are usually written in the so-called "dotted decimal" form, such as `64.89.144.135`, where each of the four numbers in the address represents an 8-bit integer in the range 0 through 255. A new version of the Internet Protocol, ***IPv6***, is currently being introduced. IPv6 addresses are 128-bit integers and are usually written in hexadecimal form (with some colons and maybe some extra information thrown in). In actual use, IPv6 addresses are still fairly rare.

A computer can have several IP addresses, and can have both IPv4 and IPv6 addresses. Usually, one of these is the ***loopback address***, which can be used when a program wants to communicate with another program *on the same computer*. The loopback address has IPv4 address 127.0.0.1 and can also, in general, be referred to using the domain name ***localhost***. In addition, there can be one or more IP addresses associated with physical network connections. Your computer probably has some utility for displaying your computer's IP addresses. I have written a small Java program, *ShowMyNetwork.java*, that does the same thing. When I run `ShowMyNetwork` on my computer, the output is:

```
en1 :  /192.168.1.47  /fe80:0:0:0:211:24ff:fe9c:5271%5
lo0 :  /127.0.0.1  /fe80:0:0:0:0:0:0:1%1  /0:0:0:0:0:0:0:1%0
```

The first thing on each line is a network interface name, which is really meaningful only to the computer's operating system. The output also contains the IP addresses for that interface. In this example, `lo0` refers to the loopback address, which has IPv4 address 127.0.0.1 as usual. The most important number here is `192.168.1.47`, which is the IPv4 address that can be used for communication over the network. The other numbers in the output are IPv6 addresses.

Now, a single computer might have several programs doing network communication at the same time, or one program communicating with several other computers. To allow for this possibility, a network connection is actually identified by a ***port number*** in combination with an IP address. A port number is just a 16-bit integer. A server does not simply listen for connections—it listens for connections *on a particular port*. A potential client must know both the Internet address (or domain name) of the computer on which the server is running and the port number on which the server is listening. A Web server, for example, generally listens for connections on port 80; other standard Internet services also have standard port numbers. (The standard port numbers are all less than 1024, and are reserved for particular services. If you create your own server programs, you should use port numbers greater than 1024.)

11.4.3 Sockets in Java

To implement TCP/IP connections, the `java.net` package provides two classes, *ServerSocket* and *Socket*. A *ServerSocket* represents a listening socket that waits for connection requests

from clients. A *Socket* represents one endpoint of an actual network connection. A *Socket* can be a client socket that sends a connection request to a server. But a *Socket* can also be created by a server to handle a connection request from a client. This allows the server to create multiple sockets and handle multiple connections. A *ServerSocket* does not itself participate in connections; it just listens for connection requests and creates *Sockets* to handle the actual connections.

When you construct a `ServerSocket` object, you have to specify the port number on which the server will listen. The specification for the constructor is

```
public ServerSocket(int port) throws IOException
```

The port number must be in the range 0 through 65535, and should generally be greater than 1024. The constructor might throw a *SecurityException* if a smaller port number is specified. An *IOException* can occur if, for example, the specified port number is already in use. (A parameter value of 0 in this method tells the server socket to listen on any available port.)

As soon as a *ServerSocket* is created, it starts listening for connection requests. The `accept()` method in the *ServerSocket* class accepts such a request, establishes a connection with the client, and returns a *Socket* that can be used for communication with the client. The `accept()` method has the form

```
public Socket accept() throws IOException
```

When you call the `accept()` method, it will not return until a connection request is received (or until some error occurs). The method is said to ***block*** while waiting for the connection. (While the method is blocked, the program—or more exactly, the thread—that called the method can't do anything else. If there are other threads in the same program, they can proceed.) You can call `accept()` repeatedly to accept multiple connection requests. The *ServerSocket* will continue listening for connections until it is closed, using its `close()` method, or until some error occurs, or until the program is terminated in some way.

Suppose that you want a server to listen on port 1728, and suppose that you've written a method `provideService(Socket)` to handle the communication with one client. Then the basic form of the server program would be:

```
try {
   ServerSocket server = new ServerSocket(1728);
   while (true) {
      Socket connection = server.accept();
      provideService(connection);
   }
}
catch (IOException e) {
   System.out.println("Server shut down with error: " + e);
}
```

On the client side, a client socket is created using a constructor in the *Socket* class. To connect to a server on a known computer and port, you would use the constructor

```
public Socket(String computer, int port) throws IOException
```

The first parameter can be either an IP number or a domain name. This constructor will block until the connection is established or until an error occurs.

Once you have a connected socket, no matter how it was created, you can use the *Socket* methods `getInputStream()` and `getOutputStream()` to obtain streams that can be used for communication over the connection. These methods return objects of type *InputStream* and

OutputStream, respectively. Keeping all this in mind, here is the outline of a method for working with a client connection:

```
/**
 * Open a client connection to a specified server computer and
 * port number on the server, and then do communication through
 * the connection.
 */
void doClientConnection(String computerName, int serverPort) {
   Socket connection;
   InputStream in;
   OutputStream out;
   try {
      connection = new Socket(computerName,serverPort);
      in = connection.getInputStream();
      out = connection.getOutputStream();
   }
   catch (IOException e) {
      System.out.println(
          "Attempt to create connection failed with error: " + e);
      return;
   }
    .
    .  // Use the streams, in and out, to communicate with the server.
    .
   try {
      connection.close();
          // (Alternatively, you might depend on the server
          //  to close the connection.)
   }
   catch (IOException e) {
   }
}  // end doClientConnection()
```

All this makes network communication sound easier than it really is. (And if you think it sounded hard, then it's even harder.) If networks were completely reliable, things would be almost as easy as I've described. The problem, though, is to write robust programs that can deal with network and human error. I won't go into detail here. However, what I've covered here should give you the basic ideas of network programming, and it is enough to write some simple network applications. Let's look at a few working examples of client/server programming.

11.4.4 A Trivial Client/Server

The first example consists of two programs. The source code files for the programs are *DateClient.java* and *DateServer.java*. One is a simple network client and the other is a matching server. The client makes a connection to the server, reads one line of text from the server, and displays that text on the screen. The text sent by the server consists of the current date and time on the computer where the server is running. In order to open a connection, the client must know the computer on which the server is running and the port on which it is listening. The server listens on port number 32007. The port number could be anything between 1025 and 65535, as long as the server and the client use the same port. Port numbers between 1 and 1024 are reserved for standard services and should not be used for other servers. The name or

IP number of the computer on which the server is running must be specified as a command-line argument. For example, if the server is running on a computer named math.hws.edu, then you would typically run the client with the command "`java DateClient math.hws.edu`". Here is the complete client program:

```
import java.net.*;
import java.io.*;

/**
 * This program opens a connection to a computer specified
 * as the first command-line argument.  The connection is made to
 * the port specified by LISTENING_PORT.  The program reads one
 * line of text from the connection and then closes the
 * connection.  It displays the text that it read on
 * standard output.  This program is meant to be used with
 * the server program, DateServer, which sends the current
 * date and time on the computer where the server is running.
 */
public class DateClient {

   public static final int LISTENING_PORT = 32007;

   public static void main(String[] args) {

      String hostName;         // Name of the server computer to connect to.
      Socket connection;       // A socket for communicating with the server.
      BufferedReader incoming; // For reading data from the connection.

      /* Get computer name from command line. */

      if (args.length > 0)
         hostName = args[0];
      else {
            // No computer name was given.  Print a message and exit.
         System.out.println("Usage:  java DateClient <server_host_name>");
         return;
      }

      /* Make the connection, then read and display a line of text. */

      try {
         connection = new Socket( hostName, LISTENING_PORT );
         incoming = new BufferedReader(
                         new InputStreamReader(connection.getInputStream()) );
         String lineFromServer = incoming.readLine();
         if (lineFromServer == null) {
               // A null from incoming.readLine() indicates that
               // end-of-stream was encountered.
            throw new IOException("Connection was opened, " +
                  "but server did not send any data.");
         }
         System.out.println();
         System.out.println(lineFromServer);
         System.out.println();
         incoming.close();
      }
      catch (Exception e) {
```

```
            System.out.println("Error:  " + e);
         }

      }  // end main()

   } //end class DateClient
```

Note that all the communication with the server is done in a `try..catch` statement. This will catch the *IOExceptions* that can be generated when the connection is opened or closed and when data is read from the input stream. The connection's input stream is wrapped in a *BufferedReader*, which has a `readLine()` method that makes it easy to read one line of text. (See Subsection 11.1.4.)

In order for this program to run without error, the server program must be running on the computer to which the client tries to connect. By the way, it's possible to run the client and the server program on the same computer. For example, you can open two command windows, start the server in one window and then run the client in the other window. To make things like this easier, most computers will recognize the domain name `localhost` and the IP number `127.0.0.1` as referring to "this computer." This means that the command "`java DateClient localhost`" will tell the *DateClient* program to connect to a server running on the same computer. If that command doesn't work, try "`java DateClient 127.0.0.1`".

The server program that corresponds to the *DateClient* client program is called *DateServer*. The *DateServer* program creates a *ServerSocket* to listen for connection requests on port 32007. After the listening socket is created, the server will enter an infinite loop in which it accepts and processes connections. This will continue until the program is killed in some way—for example by typing a `CONTROL-C` in the command window where the server is running. When a connection request is received from a client, the server calls a subroutine to handle the connection. In the subroutine, any *Exception* that occurs is caught, so that it will not crash the server. Just because a connection to one client has failed for some reason, it does not mean that the server should be shut down; the error might have been the fault of the client. The connection-handling subroutine creates a *PrintWriter* for sending data over the connection. It writes the current date and time to this stream and then closes the connection. (The standard class `java.util.Date` is used to obtain the current time. An object of type *Date* represents a particular date and time. The default constructor, "`new Date()`", creates an object that represents the time when the object is created.) The complete server program is as follows:

```
import java.net.*;
import java.io.*;
import java.util.Date;

/**
 * This program is a server that takes connection requests on
 * the port specified by the constant LISTENING_PORT.  When a
 * connection is opened, the program sends the current time to
 * the connected socket.  The program will continue to receive
 * and process connections until it is killed (by a CONTROL-C,
 * for example).  Note that this server processes each connection
 * as it is received, rather than creating a separate thread
 * to process the connection.
 */
public class DateServer {

   public static final int LISTENING_PORT = 32007;
```

```
   public static void main(String[] args) {

      ServerSocket listener;  // Listens for incoming connections.
      Socket connection;      // For communication with the connecting program.

      /* Accept and process connections forever, or until some error occurs.
         (Note that errors that occur while communicating with a connected
         program are caught and handled in the sendDate() routine, so
         they will not crash the server.) */

      try {
         listener = new ServerSocket(LISTENING_PORT);
         System.out.println("Listening on port " + LISTENING_PORT);
         while (true) {
                // Accept next connection request and handle it.
            connection = listener.accept();
            sendDate(connection);
         }
      }
      catch (Exception e) {
         System.out.println("Sorry, the server has shut down.");
         System.out.println("Error:  " + e);
         return;
      }

   }  // end main()


   /**
    * The parameter, client, is a socket that is already connected to another
    * program.  Get an output stream for the connection, send the current time,
    * and close the connection.
    */
   private static void sendDate(Socket client) {
      try {
         System.out.println("Connection from " +
                                      client.getInetAddress().toString() );
         Date now = new Date();  // The current date and time.
         PrintWriter outgoing;   // Stream for sending data.
         outgoing = new PrintWriter( client.getOutputStream() );
         outgoing.println( now.toString() );
         outgoing.flush();  // Make sure the data is actually sent!
         client.close();
      }
      catch (Exception e){
         System.out.println("Error: " + e);
      }
   } // end sendDate()

} //end class DateServer
```

When you run *DateServer* in a command-line interface, it will sit and wait for connection requests and report them as they are received. To make the *DateServer* service permanently available on a computer, the program really should be run as a ***daemon***. A daemon is a program that runs continually on a computer, independently of any user. The computer can be configured to start the daemon automatically as soon as the computer boots up. It then runs

in the background, even while the computer is being used for other purposes. For example, a computer that makes pages available on the World Wide Web runs a daemon that listens for requests for web pages and responds by transmitting the pages. It's just a souped-up analog of the *DateServer* program! However, the question of how to set up a program as a daemon is not one I want to go into here. For testing purposes, it's easy enough to start the program by hand, and, in any case, my examples are not really robust enough or full-featured enough to be run as serious servers. (By the way, the word "daemon" is just an alternative spelling of "demon" and is usually pronounced the same way.)

Note that after calling `outgoing.println()` to send a line of data to the client, the server program calls `outgoing.flush()`. The `flush()` method is available in every output stream class. Calling it ensures that data that has been written to the stream is actually sent to its destination. You should generally call this function every time you use an output stream to send data over a network connection. If you don't do so, it's possible that the stream will collect data until it has a large batch of data to send. This is done for efficiency, but it can impose unacceptable delays when the client is waiting for the transmission. It is even possible that some of the data might remain untransmitted when the socket is closed, so it is especially important to call `flush()` before closing the connection. This is one of those unfortunate cases where different implementations of Java can behave differently. If you fail to flush your output streams, it is possible that your network application will work on some types of computers but not on others.

11.4.5 A Simple Network Chat

In the *DateServer* example, the server transmits information and the client reads it. It's also possible to have two-way communication between client and server. As a first example, we'll look at a client and server that allow a user on each end of the connection to send messages to the other user. The program works in a command-line interface where the users type in their messages. In this example, the server waits for a connection from a single client and then closes down its listener so that no other clients can connect. After the client and server are connected, both ends of the connection work in much the same way. The user on the client end types a message, and it is transmitted to the server, which displays it to the user on that end. Then the user of the server types a message that is transmitted to the client. Then the client user types another message, and so on. This continues until one user or the other enters "quit" when prompted for a message. When that happens, the connection is closed and both programs terminate. The client program and the server program are very similar. The techniques for opening the connections differ, and the client is programmed to send the first message while the server is programmed to receive the first message. The client and server programs can be found in the files *CLChatClient.java* and *CLChatServer.java*. (The name "CLChat" stands for "command-line chat.") Here is the source code for the server; the client is similar:

```
import java.net.*;
import java.io.*;

/**
 * This program is one end of a simple command-line interface chat program.
 * It acts as a server which waits for a connection from the CLChatClient
 * program.  The port on which the server listens can be specified as a
 * command-line argument.  If it is not, then the port specified by the
 * constant DEFAULT_PORT is used.  Note that if a port number of zero is
 * specified, then the server will listen on any available port.
```

```
 * This program only supports one connection.  As soon as a connection is
 * opened, the listening socket is closed down.  The two ends of the connection
 * each send a HANDSHAKE string to the other, so that both ends can verify
 * that the program on the other end is of the right type.  Then the connected
 * programs alternate sending messages to each other.  The client always sends
 * the first message.  The user on either end can close the connection by
 * entering the string "quit" when prompted for a message.  Note that the first
 * character of any string sent over the connection must be 0 or 1; this
 * character is interpreted as a command.
 */
public class CLChatServer {

   /**
    * Port to listen on, if none is specified on the command line.
    */
   static final int DEFAULT_PORT = 1728;

   /**
    * Handshake string. Each end of the connection sends this string to the
    * other just after the connection is opened.  This is done to confirm that
    * the program on the other side of the connection is a CLChat program.
    */
   static final String HANDSHAKE = "CLChat";

   /**
    * This character is prepended to every message that is sent.
    */
   static final char MESSAGE = '0';

   /**
    * This character is sent to the connected program when the user quits.
    */
   static final char CLOSE = '1';


   public static void main(String[] args) {

      int port;   // The port on which the server listens.

      ServerSocket listener;  // Listens for a connection request.
      Socket connection;      // For communication with the client.

      BufferedReader incoming;  // Stream for receiving data from client.
      PrintWriter outgoing;     // Stream for sending data to client.
      String messageOut;        // A message to be sent to the client.
      String messageIn;         // A message received from the client.

      BufferedReader userInput; // A wrapper for System.in, for reading
                                // lines of input from the user.

      /* First, get the port number from the command line,
         or use the default port if none is specified. */

      if (args.length == 0)
         port = DEFAULT_PORT;
      else {
         try {
            port= Integer.parseInt(args[0]);
            if (port < 0 || port > 65535)
```

```
            throw new NumberFormatException();
      }
      catch (NumberFormatException e) {
         System.out.println("Illegal port number, " + args[0]);
         return;
      }
   }

   /* Wait for a connection request.  When it arrives, close
      down the listener.  Create streams for communication
      and exchange the handshake. */

   try {
      listener = new ServerSocket(port);
      System.out.println("Listening on port " + listener.getLocalPort());
      connection = listener.accept();
      listener.close();
      incoming = new BufferedReader(
                       new InputStreamReader(connection.getInputStream()) );
      outgoing = new PrintWriter(connection.getOutputStream());
      outgoing.println(HANDSHAKE);  // Send handshake to client.
      outgoing.flush();  // Make sure handshake is transmitted NOW.
      messageIn = incoming.readLine();  // Receive handshake from client.
      if (! HANDSHAKE.equals(messageIn) ) {
         throw new Exception("Connected program is not a CLChat!");
      }
      System.out.println("Connected.  Waiting for the first message.");
   }
   catch (Exception e) {
      System.out.println("An error occurred while opening connection.");
      System.out.println(e.toString());
      return;
   }

   /* Exchange messages with the other end of the connection until one side
      or the other closes the connection.  This server program waits for
      the first message from the client.  After that, messages alternate
      strictly back and forth. */

   try {
      userInput = new BufferedReader(new InputStreamReader(System.in));
      System.out.println("NOTE: Enter 'quit' to end the program.\n");
      while (true) {
         System.out.println("WAITING...");
         messageIn = incoming.readLine();
         if (messageIn.length() > 0) {
                // The first character of the message is a command. If
                // the command is CLOSE, then the connection is closed.
                // Otherwise, remove the command character from the
                // message and proceed.
            if (messageIn.charAt(0) == CLOSE) {
               System.out.println("Connection closed at other end.");
               connection.close();
               break;
            }
```

```
                messageIn = messageIn.substring(1);
             }
             System.out.println("RECEIVED:  " + messageIn);
             System.out.print("SEND:      ");
             messageOut = userInput.readLine();
             if (messageOut.equalsIgnoreCase("quit"))  {
                   // User wants to quit.  Inform the other side
                   // of the connection, then close the connection.
                outgoing.println(CLOSE);
                outgoing.flush();  // Make sure the data is sent!
                connection.close();
                System.out.println("Connection closed.");
                break;
             }
             outgoing.println(MESSAGE + messageOut);
             outgoing.flush(); // Make sure the data is sent!
             if (outgoing.checkError()) {
                throw new IOException("Error occurred while transmitting message.");
             }
          }
       }
       catch (Exception e) {
          System.out.println("Sorry, an error has occurred.  Connection lost.");
          System.out.println("Error:  " + e);
          System.exit(1);
       }

    }  // end main()

} //end class CLChatServer
```

This program is a little more robust than *DateServer*. For one thing, it uses a ***handshake*** to make sure that a client who is trying to connect is really a *CLChatClient* program. A handshake is simply information sent between a client and a server as part of setting up a connection, before any actual data is sent. In this case, each side of the connection sends a string to the other side to identify itself. The handshake is part of the ***protocol*** that I made up for communication between *CLChatClient* and *CLChatServer*. A protocol is a detailed specification of what data and messages can be exchanged over a connection, how they must be represented, and what order they can be sent in. When you design a client/server application, the design of the protocol is an important consideration. Another aspect of the `CLChat` protocol is that after the handshake, every line of text that is sent over the connection begins with a character that acts as a command. If the character is 0, the rest of the line is a message from one user to the other. If the character is 1, the line indicates that a user has entered the "quit" command, and the connection is to be shut down.

Remember that if you want to try out this program on a single computer, you can use two command-line windows. In one, give the command "`java CLChatServer`" to start the server. Then, in the other, use the command "`java CLChatClient localhost`" to connect to the server that is running on the same machine.

11.5 A Brief Introduction to XML

When data is saved to a file or transmitted over a network, it must be represented in some way that will allow the same data to be rebuilt later, when the file is read or the transmission is received. We have seen that there are good reasons to prefer textual, character-based representations in many cases, but there are many ways to represent a given collection of data as text. In this section, we'll take a brief look at one type of character-based data representation that has become increasingly common.

XML (eXtensible Markup Language) is a syntax for creating data representation languages. There are two aspects or levels of XML. On the first level, XML specifies a strict but relatively simple syntax. Any sequence of characters that follows that syntax is a ***well-formed*** XML document. On the second level, XML provides a way of placing further restrictions on what can appear in a document. This is done by associating a ***DTD*** (Document Type Definition) with an XML document. A DTD is essentially a list of things that are allowed to appear in the XML document. A well-formed XML document that has an associated DTD and that follows the rules of the DTD is said to be a ***valid*** XML document. The idea is that XML is a general format for data representation, and a DTD specifies how to use XML to represent a particular kind of data. (There is also an alternative to DTDs, known as ***XML schemas***, for defining valid XLM documents, but let's ignore them here.)

There is nothing magical about XML. It's certainly not perfect. It's a very verbose language, and some people think it's ugly. On the other hand it's very flexible; it can be used to represent almost any type of data. It was built from the start to support all languages and alphabets. Most important, it has become an accepted standard. There is support in just about any programming language for processing XML documents. There are standard DTDs for describing many different kinds of data. There are many ways to design a data representation language, but XML is the one that has happened to come into widespread use. In fact, it has found its way into almost every corner of information technology. For example: There are XML languages for representing mathematical expressions (MathML), musical notation (MusicXML), molecules and chemical reactions (CML), vector graphics (SVG), and many other kinds of information. XML is used by OpenOffice and recent versions of Microsoft Office in the document format for office applications such as word processing, spreadsheets, and presentations. XML site syndication languages (RSS, ATOM) make it possible for web sites, newspapers, and blogs to make a list of recent headlines available in a standard format that can be used by other web sites and by web browsers; the same format is used to publish podcasts. And XML is a common format for the electronic exchange of business information.

My purpose here is not to tell you everything there is to know about XML. I will just explain a few ways in which it can be used in your own programs. In particular, I will not say anything further about DTDs and valid XML. For many purposes, it is sufficient to use well-formed XML documents with no associated DTDs.

11.5.1 Basic XML Syntax

An XML document looks a lot like an HTML document (see Subsection 6.2.3). HTML is not itself an XML language, since it does not follow all the strict XML syntax rules, but the basic ideas are similar. Here is a short, well-formed XML document:

```
<?xml version="1.0"?>
<simplepaint version="1.0">
   <background red='255' green='153' blue='51'/>
```

```
   <curve>
      <color red='0' green='0' blue='255'/>
      <symmetric>false</symmetric>
      <point x='83' y='96'/>
      <point x='116' y='149'/>
      <point x='159' y='215'/>
      <point x='216' y='294'/>
      <point x='264' y='359'/>
      <point x='309' y='418'/>
      <point x='371' y='499'/>
      <point x='400' y='543'/>
   </curve>
   <curve>
      <color red='255' green='255' blue='255'/>
      <symmetric>true</symmetric>
      <point x='54' y='305'/>
      <point x='79' y='289'/>
      <point x='128' y='262'/>
      <point x='190' y='236'/>
      <point x='253' y='209'/>
      <point x='341' y='158'/>
   </curve>
</simplepaint>
```

The first line, which is optional, merely identifies this as an XML document. This line can also specify other information, such as the character encoding that was used to encode the characters in the document into binary form. If this document had an associated DTD, it would be specified in a "DOCTYPE" directive on the next line of the file.

Aside from the first line, the document is made up of ***elements***, ***attributes***, and textual content. An element starts with a ***tag***, such as `<curve>` and ends with a matching ***end-tag*** such as `</curve>`. Between the tag and end-tag is the ***content*** of the element, which can consist of text and nested elements. (In the example, the only textual content is the `true` or `false` in the `<symmetric>` elements.) If an element has no content, then the opening tag and end-tag can be combined into a single ***empty tag***, such as `<point x='83' y='96'/>`, with a "/" before the final ">". This is an abbreviation for `<point x='83' y='96'></point>`. A tag can include attributes such as the `x` and `y` in `<point x='83' y='96'/>` or the `version` in `<simplepaint version="1.0">`. A document can also include a few other things, such as comments, that I will not discuss here.

The basic structure should look familiar to someone familiar with HTML. The most striking difference is that in XML, **you get to choose the tags**. Whereas HTML comes with a fixed, finite set of tags, with XML you can make up meaningful tag names that are appropriate to your application and that describe the data that is being represented. (For an XML document that uses a DTD, it's the author of the DTD who gets to choose the tag names.)

Every well-formed XML document follows a strict syntax. Here are some of the most important syntax rules: Tag names and attribute names in XML are case sensitive. A name must begin with a letter and can contain letters, digits and certain other characters. Spaces and ends-of-line are significant only in textual content. Every tag must either be an empty tag or have a matching end-tag. By "matching" here, I mean that elements must be properly nested; if a tag is inside some element, then the matching end-tag must also be inside that element. A document must have a ***root element***, which contains all the other elements. The root element in the above example has tag name `simplepaint`. Every attribute must have

a value, and that value must be enclosed in quotation marks; either single quotes or double quotes can be used for this. The special characters < and &, if they appear in attribute values or textual content, must be written as `<` and `&`. "`<`" and "`&`" are examples of ***entities***. The entities `>`, `&quot;`, and `&apos;` are also defined, representing >, double quote, and single quote. (Additional entities can be defined in a DTD.)

While this description will not enable you to understand everything that you might encounter in XML documents, it should allow you to design well-formed XML documents to represent data structures used in Java programs.

11.5.2 XMLEncoder and XMLDecoder

We will look at two approaches to representing data from Java programs in XML format. One approach is to design a custom XML language for the specific data structures that you want to represent. We will consider this approach in the next subsection. First, we'll look at an easy way to store data in XML files and to read those files back into a program. The technique uses the classes *XMLEncoder* and *XMLDecoder*. These classes are defined in the package `java.beans`. An *XMLEncoder* can be used to write objects to an *OutputStream* in XML form. An *XMLDecoder* can be used to read the output of an *XMLEncoder* and reconstruct the objects that were written by it. *XMLEncoder* and *XMLDecoder* have much the same functionality as *ObjectOutputStream* and *ObjectInputStream* and are used in much the same way. In fact, you don't even have to know anything about XML to use them. However, you do need to know a little about ***Java beans***.

A Java bean is just an object that has certain characteristics. The class that defines a Java bean must be a `public` class. It must have a constructor that takes no parameters. It should have a "get" method and a "set" method for each of its important instance variables. (See Subsection 5.1.3.) The last rule is a little vague. The idea is that it should be possible to inspect all aspects of the object's state by calling "get" methods, and it should be possible to set all aspects of the state by calling "set" methods. A bean is not required to implement any particular `interface`; it is recognized as a bean just by having the right characteristics. Usually, Java beans are passive data structures that are acted upon by other objects but don't do much themselves.

XMLEncoder and *XMLDecoder* can't be used with arbitrary objects; they can only be used with beans. When an *XMLEncoder* writes an object, it uses the "get" methods of that object to find out what information needs to be saved. When an *XMLDecoder* reconstructs an object, it creates the object using the constructor with no parameters and it uses "set" methods to restore the object's state to the values that were saved by the *XMLEncoder*. (Some standard Java classes are processed using additional techniques. For example, a different constructor might be used, and other methods might be used to inspect and restore the state.)

For an example, we return to the same `SimplePaint` example that was used in Subsection 11.3.4. Suppose that we want to use *XMLEncoder* and *XMLDecoder* to create and read files in that program. Part of the data for a `SimplePaint` sketch is stored in objects of type *CurveData*, defined as:

```
private static class CurveData {
   Color color;  // The color of the curve.
   boolean symmetric;  // Are reflections also drawn?
   ArrayList<Point> points;  // The points on the curve.
}
```

To use such objects with *XMLEncoder* and *XMLDecoder*, we have to modify this class so that it follows the Java bean pattern. The class has to be public, and we need get and set methods for each instance variable. This gives:

```
public static class CurveData {
   private Color color;  // The color of the curve.
   private boolean symmetric;  // Are reflections also drawn?
   private ArrayList<Point> points;  // The points on the curve.
   public Color getColor() {
      return color;
   }
   public void setColor(Color color) {
      this.color = color;
   }
   public ArrayList<Point> getPoints() {
      return points;
   }
   public void setPoints(ArrayList<Point> points) {
      this.points = points;
   }
   public boolean isSymmetric() {
      return symmetric;
   }
   public void setSymmetric(boolean symmetric) {
      this.symmetric = symmetric;
   }
}
```

I didn't really need to make the instance variables `private`, but bean properties are usually `private` and are accessed only through their get and set methods.

At this point, we might define another bean class, *SketchData*, to hold all the necessary data for representing the user's picture. If we did that, we could write the data to a file with a single output statement. In my program, however, I decided to write the data in several pieces.

An *XMLEncoder* can be constructed to write to any output stream. The output stream is specified in the encoder's constructor. For example, to create an encoder for writing to a file:

```
XMLEncoder encoder;
try {
   FileOutputStream stream = new FileOutputStream(selectedFile);
   encoder = new XMLEncoder( stream );
     .
     .
```

Once an encoder has been created, its `writeObject()` method is used to write objects, coded into XML form, to the stream. In the `SimplePaint` program, I save the background color, the number of curves in the picture, and the data for each curve. The curve data are stored in a list of type *ArrayList<CurveData>* named `curves`. So, a complete representation of the user's picture can be created with:

```
   encoder.writeObject(getBackground());
   encoder.writeObject(new Integer(curves.size()));
   for (CurveData c : curves)
      encoder.writeObject(c);
   encoder.close();
```

When reading the data back into the program, an *XMLDecoder* is created to read from an input file stream. The objects are then read, using the decoder's `readObject()` method, in the same order in which they were written. Since the return type of `readObject()` is *Object*, the returned values must be type-cast to their correct type:

```
Color bgColor = (Color)decoder.readObject();
Integer curveCt = (Integer)decoder.readObject();
ArrayList<CurveData> newCurves = new ArrayList<CurveData>();
for (int i = 0; i < curveCt; i++) {
   CurveData c = (CurveData)decoder.readObject();
   newCurves.add(c);
}
decoder.close();
curves = newCurves; // Replace the program's data with data from the file.
setBackground(bgColor);
repaint();
```

You can look at the sample program *SimplePaintWithXMLEncoder.java* to see this code in the context of a complete program. Files are created by the method `doSaveAsXML()` and are read by `doOpenAsXML()`.

The XML format used by *XMLEncoder* and *XMLDecoder* is more robust than the binary format used for object streams and is more appropriate for long-term storage of objects in files.

11.5.3 Working With the DOM

The output produced by an *XMLEncoder* tends to be long and not very easy for a human reader to understand. It would be nice to represent data in a more compact XML format that uses meaningful tag names to describe the data and makes more sense to human readers. We'll look at yet another version of `SimplePaint` that does just that. See *SimplePaintWithXML.java* for the source code. The sample XML document shown earlier in this section was produced by this program. I designed the format of that document to represent all the data needed to reconstruct a picture in `SimplePaint`. The document encodes the background color of the picture and a list of curves. Each `<curve>` element contains the data from one object of type *CurveData*.

It is easy enough to write data in a customized XML format, although we have to be very careful to follow all the syntax rules. Here is how I write the data for a `SimplePaint` picture to a *PrintWriter*, `out`:

```
out.println("<?xml version=\"1.0\"?>");
out.println("<simplepaint version=\"1.0\">");
Color bgColor = getBackground();
out.println("   <background red='" + bgColor.getRed() + "' green='" +
      bgColor.getGreen() + "' blue='" + bgColor.getBlue() + "'/>");
for (CurveData c : curves) {
   out.println("   <curve>");
   out.println("      <color red='" + c.color.getRed() + "' green='" +
         c.color.getGreen() + "' blue='" + c.color.getBlue() + "'/>");
   out.println("      <symmetric>" + c.symmetric + "</symmetric>");
   for (Point pt : c.points)
      out.println("      <point x='" + pt.x + "' y='" + pt.y + "'/>");
   out.println("   </curve>");
}
out.println("</simplepaint>");
```

Reading the data back into the program is another matter. To reconstruct the data structure represented by the XML Document, it is necessary to parse the document and extract the data from it. This could be difficult to do by hand. Fortunately, Java has a standard API for parsing and processing XML Documents. (Actually, it has two, but we will only look at one of them.)

A well-formed XML document has a certain structure, consisting of elements containing attributes, nested elements, and textual content. It's possible to build a data structure in the computer's memory that corresponds to the structure and content of the document. Of course, there are many ways to do this, but there is one common standard representation known as the ***Document Object Model***, or DOM. The DOM specifies how to build data structures to represent XML documents, and it specifies some standard methods for accessing the data in that structure. The data structure is a kind of tree whose structure mirrors the structure of the document. The tree is constructed from ***nodes*** of various types. There are nodes to represent elements, attributes, and text. (The tree can also contain several other types of node, representing aspects of XML that we can ignore here.) Attributes and text can be processed without directly manipulating the corresponding nodes, so we will be concerned almost entirely with element nodes.

The sample program *XMLDemo.java* lets you experiment with XML and the DOM. It has a text area where you can enter an XML document. Initially, the input area contains the sample XML document from this section. When you click a button named "Parse XML Input", the program will attempt to read the XML from the input box and build a DOM representation of that document. If the input is not legal XML, an error message is displayed. If it is legal, the program will traverse the DOM representation and display a list of elements, attributes, and textual content that it encounters. (The program uses a few techniques that I won't discuss here.)

In Java, the DOM representation of an XML document file can be created with just two statements. If `selectedFile` is a variable of type *File* that represents the XML file, then

```
DocumentBuilder docReader
                = DocumentBuilderFactory.newInstance().newDocumentBuilder();
xmldoc = docReader.parse(selectedFile);
```

will open the file, read its contents, and build the DOM representation. The classes *DocumentBuilder* and *DocumentBuilderFactory* are both defined in the package `javax.xml.parsers`. The method `docReader.parse()` does the actual work. It will throw an exception if it can't read the file or if the file does not contain a legal XML document. If it succeeds, then the value returned by `docReader.parse()` is an object that represents the entire XML document. (This is a very complex task! It has been coded once and for all into a method that can be used very easily in any Java program. We see the benefit of using a standardized syntax.)

The structure of the DOM data structure is defined in the package `org.w3c.dom`, which contains several data types that represent an XML document as a whole and the individual nodes in a document. The "org.w3c" in the name refers to the World Wide Web Consortium, W3C, which is the standards organization for the Web. DOM, like XML, is a general standard, not just a Java standard. The data types that we need here are *Document*, *Node*, *Element*, and *NodeList*. (They are defined as `interfaces` rather than `classes`, but that fact is not relevant here.) We can use methods that are defined in these data types to access the data in the DOM representation of an XML document.

An object of type *Document* represents an entire XML document. The return value of `docReader.parse()`—`xmldoc` in the above example—is of type *Document*. We will only need one method from this class: If `xmldoc` is of type *Document*, then

```
xmldoc.getDocumentElement()
```

returns a value of type *Element* that represents the root element of the document. (Recall that this is the top-level element that contains all the other elements.) In the sample XML document from earlier in this section, the root element consists of the tag `<simplepaint version="1.0">`, the end-tag `</simplepaint>`, and everything in between. The elements that are nested inside the root element are represented by their own nodes, which are said to be ***children*** of the root node. An object of type *Element* contains several useful methods. If `element` is of type *Element*, then we have:

- `element.getTagName()` — returns a *String* containing the name that is used in the element's tag. For example, the name of a `<curve>` element is the string "curve".
- `element.getAttribute(attrName)` — if `attrName` is the name of an attribute in the element, then this method returns the value of that attribute. For the element, `<point x="83" y="42"/>`, `element.getAttribute("x")` would return the string "83". Note that the return value is always a *String*, even if the attribute is supposed to represent a numerical value. If the element has no attribute with the specified name, then the return value is an empty string.
- `element.getTextContent()` — returns a *String* containing all the textual content that is contained in the element. Note that this includes text that is contained inside other elements that are nested inside the element.
- `element.getChildNodes()` — returns a value of type *NodeList* that contains all the *Nodes* that are children of the element. The list includes nodes representing other elements and textual content that are directly nested in the element (as well as some other types of node that I don't care about here). The `getChildNodes()` method makes it possible to traverse the entire DOM data structure by starting with the root element, looking at children of the root element, children of the children, and so on. (There is a similar method that returns the attributes of the element, but I won't be using it here.)
- `element.getElementsByTagName(tagName)` — returns a *NodeList* that contains all the nodes representing all elements that are nested inside `element` and which have the given tag name. Note that this includes elements that are nested to any level, not just elements that are directly contained inside `element`. The `getElementsByTagName()` method allows you to reach into the document and pull out specific data that you are interested in.

An object of type *NodeList* represents a list of *Nodes*. Unfortunately, it does not use the API defined for lists in the Java Collection Framework. Instead, a value, `nodeList`, of type *NodeList* has two methods: `nodeList.getLength()` returns the number of nodes in the list, and `nodeList.item(i)` returns the node at position `i`, where the positions are numbered `0`, `1`, ..., `nodeList.getLength() - 1`. Note that the return value of `nodeList.get()` is of type *Node*, and it might have to be type-cast to a more specific node type before it is used.

Knowing just this much, you can do the most common types of processing of DOM representations. Let's look at a few code fragments. Suppose that in the course of processing a document you come across an *Element* node that represents the element

```
&lt;background red='255' green='153' blue='51'/&gt;
```

This element might be encountered either while traversing the document with `getChildNodes()` or in the result of a call to `getElementsByTagName("background")`. Our goal is to reconstruct the data structure represented by the document, and this element represents part of that data. In this case, the element represents a color, and the red, green, and blue components are given

by the attributes of the element. If `element` is a variable that refers to the node, the color can be obtained by saying:

```
int r = Integer.parseInt( element.getAttribute("red") );
int g = Integer.parseInt( element.getAttribute("green") );
int b = Integer.parseInt( element.getAttribute("blue") );
Color bgColor = new Color(r,g,b);
```

Suppose now that `element` refers to the node that represents the element

```
<symmetric>true</symmetric>
```

In this case, the element represents the value of a **boolean** variable, and the value is encoded in the textual content of the element. We can recover the value from the element with:

```
String bool = element.getTextContent();
boolean symmetric;
if (bool.equals("true"))
   symmetric = true;
else
   symmetric = false;
```

Next, consider an example that uses a *NodeList*. Suppose we encounter an element that represents a list of *Points*:

```
<pointlist>
   <point x='17' y='42'/>
   <point x='23' y='8'/>
   <point x='109' y='342'/>
   <point x='18' y='270'/>
</pointlist>
```

Suppose that `element` refers to the node that represents the `<pointlist>` element. Our goal is to build the list of type `ArrayList<Point>` that is represented by the element. We can do this by traversing the *NodeList* that contains the child nodes of `element`:

```
ArrayList<Point> points = new ArrayList<Point>();
NodeList children = element.getChildNodes();
for (int i = 0; i < children.getLength(); i++) {
   Node child = children.item(i);   // One of the child nodes of element.
   if ( child instanceof Element ) {
      Element pointElement = (Element)child;  // One of the <point> elements.
      int x = Integer.parseInt( pointElement.getAttribute("x") );
      int y = Integer.parseInt( pointElement.getAttribute("y") );
      Point pt = new Point(x,y); // Create the Point represented by pointElement.
      points.add(pt);            // Add the point to the list of points.
   }
}
```

All the nested `<point>` elements are children of the `<pointlist>` element. The `if` statement in this code fragment is necessary because an element can have other children in addition to its nested elements. In this example, we only want to process the children that are elements.

All these techniques can be employed to write the file input method for the sample program *SimplePaintWithXML.java*. When building the data structure represented by an XML file, my approach is to start with a default data structure and then to modify and add to it as I traverse the DOM representation of the file. It's not a trivial process, but I hope that you can follow it:

```
Color newBackground = Color.WHITE;
ArrayList<CurveData> newCurves = new ArrayList<CurveData>();

Element rootElement = xmldoc.getDocumentElement();

if ( ! rootElement.getNodeName().equals("simplepaint") )
   throw new Exception("File is not a SimplePaint file.");
String version = rootElement.getAttribute("version");
try {
   double versionNumber = Double.parseDouble(version);
   if (versionNumber > 1.0)
      throw new Exception("File requires a newer version of SimplePaint.");
}
catch (NumberFormatException e) {
}

NodeList nodes = rootElement.getChildNodes();

for (int i = 0; i < nodes.getLength(); i++) {
   if (nodes.item(i) instanceof Element) {
      Element element = (Element)nodes.item(i);
      if (element.getTagName().equals("background")) { // Read background color.
         int r = Integer.parseInt(element.getAttribute("red"));
         int g = Integer.parseInt(element.getAttribute("green"));
         int b = Integer.parseInt(element.getAttribute("blue"));
         newBackground = new Color(r,g,b);
      }
      else if (element.getTagName().equals("curve")) { // Read data for a curve.
         CurveData curve = new CurveData();
         curve.color = Color.BLACK;
         curve.points = new ArrayList<Point>();
         newCurves.add(curve);  // Add this curve to the new list of curves.
         NodeList curveNodes = element.getChildNodes();
         for (int j = 0; j < curveNodes.getLength(); j++) {
            if (curveNodes.item(j) instanceof Element) {
               Element curveElement = (Element)curveNodes.item(j);
               if (curveElement.getTagName().equals("color")) {
                  int r = Integer.parseInt(curveElement.getAttribute("red"));
                  int g = Integer.parseInt(curveElement.getAttribute("green"));
                  int b = Integer.parseInt(curveElement.getAttribute("blue"));
                  curve.color = new Color(r,g,b);
               }
               else if (curveElement.getTagName().equals("point")) {
                  int x = Integer.parseInt(curveElement.getAttribute("x"));
                  int y = Integer.parseInt(curveElement.getAttribute("y"));
                  curve.points.add(new Point(x,y));
               }
               else if (curveElement.getTagName().equals("symmetric")) {
                  String content = curveElement.getTextContent();
                  if (content.equals("true"))
                     curve.symmetric = true;
               }
            }
         }
      }
```

```
   }
}
curves = newCurves;  // Change picture in window to show the data from file.
setBackground(newBackground);
repaint();
```

* * *

XML has developed into an extremely important technology, and some applications of it are very complex. But there is a core of simple ideas that can be easily applied in Java. Knowing just the basics, you can make good use of XML in your own Java programs.

Exercises for Chapter 11

1. The sample program *DirectoryList.java*, given as an example in Subsection 11.2.2, will print a list of files in a directory specified by the user. But some of the files in that directory might themselves be directories. And the subdirectories can themselves contain directories. And so on. Write a modified version of `DirectoryList` that will list all the files in a directory and all its subdirectories, to any level of nesting. You will need a **recursive** subroutine to do the listing. The subroutine should have a parameter of type *File*. You will need the constructor from the *File* class that has the form

```
public File( File dir, String fileName )
   // Constructs the File object representing a file
   // named fileName in the directory specified by dir.
```

2. Write a program that will count the number of lines in each file that is specified on the command line. Assume that the files are text files. Note that multiple files can be specified, as in:

```
java  LineCounts  file1.txt  file2.txt  file3.txt
```

Write each file name, along with the number of lines in that file, to standard output. If an error occurs while trying to read from one of the files, you should print an error message for that file, but you should still process all the remaining files. Do not use *TextIO* to process the files; use a *Scanner*, a *BufferedReader*, or a *TextReader* to process each file.

3. For this exercise, you will write a network server program. The program is a simple file server that makes a collection of files available for transmission to clients. When the server starts up, it needs to know the name of the directory that contains the collection of files. This information can be provided as a command-line argument. You can assume that the directory contains only regular files (that is, it does not contain any sub-directories). You can also assume that all the files are text files.

When a client connects to the server, the server first reads a one-line command from the client. The command can be the string "`index`". In this case, the server responds by sending a list of names of all the files that are available on the server. Or the command can be of the form "`get <filename>`", where `<filename>` is a file name. The server checks whether the requested file actually exists. If so, it first sends the word "ok" as a message to the client. Then it sends the contents of the file and closes the connection. Otherwise, it sends the word "error" to the client and closes the connection.

Write a subroutine to handle each request. See the `DirectoryList` example in Subsection 11.2.2 for help with the problem of getting the list of files in the directory.

4. Write a client program for the server from Exercise 11.3. Design a user interface that will let the user do at least two things: (1) Get a list of files that are available on the server and display the list on standard output; and (2) Get a copy of a specified file from the server and save it to a local file (on the computer where the client is running).

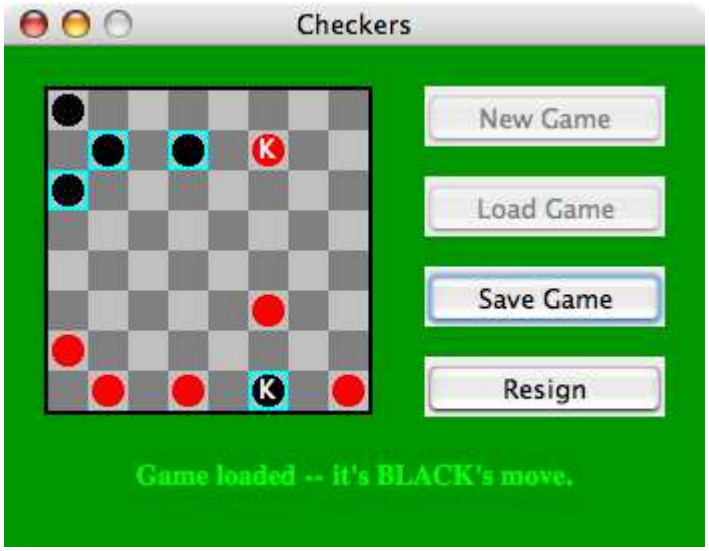
5. The sample program *PhoneDirectoryFileDemo.java*, from Subsection 11.3.2, stores name/number pairs for a simple phone book in a text file in the user's home directory. Modify that program so that it uses an XML format for the data. The only significant

changes that you will have to make are to the parts of the program that read and write the data file. Use the DOM to read the data, as discussed in Subsection 11.5.3. You can use the XML format illustrated in the following sample phone directory file:

```
<?xml version="1.0"?>
<phone_directory>
  <entry name='barney' number='890-1203'/>
  <entry name='fred' number='555-9923'/>
</phone_directory>
```

(This is just an easy exercise in simple XML processing; as before, the program in this exercise is not meant to be a useful phone directory program.)

6. The sample program *Checkers.java* from Subsection 7.5.3 lets two players play checkers. It would be nice if, in the middle of a game, the state of the game could be saved to a file. Later, the file could be read back into the file to restore the game and allow the players to continue. Add the ability to save and load files to the checkers program. Design a simple text-based format for the files. Here is a picture of my solution to this exercise, just after a file has been loaded into the program:



Note: The original checkers program could be run as either an applet or a stand-alone application. Since the new version uses files, however, it can only be run as an application. An applet running in a web browser is not allowed to access files.

It's a little tricky to completely restore the state of a game. The program has a variable `board` of type *CheckersData* that stores the current contents of the board, and it has a variable `currentPlayer` of type **int** that indicates whether Red or Black is currently moving. This data must be stored in the file when a file is saved. When a file is read into the program, you should read the data into two local variables `newBoard` of type *CheckersData* and `newCurrentPlayer` of type **int**. Once you have successfully read all the data from the file, you can use the following code to set up the program state correctly. This code assumes that you have introduced two new variables `saveButton` and `loadButton` of type *JButton* to represent the "Save Game" and "Load Game" buttons:

```
board = newBoard;  // Set up game with data read from file.
currentPlayer = newCurrentPlayer;
legalMoves = board.getLegalMoves(currentPlayer);
selectedRow = -1;
gameInProgress = true;
newGameButton.setEnabled(false);
loadButton.setEnabled(false);
saveButton.setEnabled(true);
resignButton.setEnabled(true);
if (currentPlayer == CheckersData.RED)
   message.setText("Game loaded -- it's RED's move.");
else
   message.setText("Game loaded -- it's BLACK's move.");
repaint();
```

(Note, by the way, that I used a *TextReader* to read the data from the file into my program. *TextReader* is a non-standard class introduced in Subsection 11.1.4 and defined in the file *TextReader.java*. How to read the data in a file depends, of course, on the format that you have chosen for the data.)

Quiz on Chapter 11

1. In Java, input/output is done using streams. Streams are an *abstraction.* Explain what this means and why it is important.

2. Java has two types of streams: character streams and byte streams. Why? What is the difference between the two types of streams?

3. What is a *file?* Why are files necessary?

4. What is the point of the following statement?

```
out = new PrintWriter( new FileWriter("data.dat") );
```

Why would you need a statement that involves two different stream classes, *PrintWriter* and *FileWriter*?

5. The package `java.io` includes a class named *URL*. What does an object of type *URL* represent, and how is it used?

6. What is the purpose of the *JFileChooser* class?

7. Explain what is meant by the *client / server* model of network communication.

8. What is a *socket*?

9. What is a *ServerSocket* and how is it used?

10. What is meant by an *element* in an XML document?

11. What is it about XML that makes it suitable for representing almost any type of data?

12. Write a complete program that will display the first ten lines from a text file. The lines should be written to standard output, `System.out`. The file name is given as the command-line argument `args[0]`. You can assume that the file contains at least ten lines. Don't bother to make the program robust. Do not use *TextIO* to process the file; use a *FileReader* to access the file.

Chapter 12

Threads and Multiprocessing

In the classic programming model, there is a single central processing unit that reads instructions from memory and carries them out, one after the other. The purpose of a program is to provide the list of instructions for the processor to execute. This is the only type of programming that we have considered so far.

However, this model of programming has limitations. Modern computers have multiple processors, making it possible for them to perform several tasks at the same time. To use the full potential of all those processors, you will need to write programs that can do ***parallel processing***. For Java programmers, that means learning about ***threads***. A single thread is similar to the programs that you have been writing up until now, but more than one thread can be running at the same time, "in parallel." What makes things more interesting—and more difficult—than single-threaded programming is the fact that the threads in a parallel program are rarely completely independent of one another. They usually need to cooperate and communicate. Learning to manage and control cooperation among threads is the main hurdle that you will face in this chapter.

There are several reasons to use parallel programming. One is simply to do computations more quickly by setting several processors to work on them simultaneously. Just as important, however, is to use threads to deal with "blocking" operations, where a process can't proceed until some event occurs. In the previous chapter, for example, we saw how programs can block while waiting for data to arrive over a network connection. Threads make it possible for one part of a program to continue to do useful work even while another part is blocked, waiting for some event to occur. In this context, threads are a vital programming tool even for a computer that has only a single processing unit.

12.1 Introduction to Threads

Like people, computers can ***multitask***. That is, they can be working on several different tasks at the same time. A computer that has just a single central processing unit can't literally do two things at the same time, any more than a person can, but it can still switch its attention back and forth among several tasks. Furthermore, it is increasingly common for computers to have more than one processing unit, and such computers can literally work on several tasks simultaneously. It is likely that from now on, most of the increase in computing power will come from adding additional processors to computers rather than from increasing the speed of individual processors. To use the full power of these multiprocessing computers, a programmer must do ***parallel programming***, which means writing a program as a set of several tasks that

can be executed simultaneously. Even on a single-processor computer, parallel programming techniques can be useful, since some problems can be tackled most naturally by breaking the solution into a set of simultaneous tasks that cooperate to solve the problem.

In Java, a single task is called a ***thread***. The term "thread" refers to a "thread of control" or "thread of execution," meaning a sequence of instructions that are executed one after another—the thread extends through time, connecting each instruction to the next. In a multithreaded program, there can be many threads of control, weaving through time in parallel and forming the complete fabric of the program. (Ok, enough with the metaphor, already!) Every Java program has at least one thread; when the Java virtual machine runs your program, it creates a thread that is responsible for executing the `main` routine of the program. This main thread can in turn create other threads that can continue even after the main thread has terminated. In a GUI program, there is at least one additional thread, which is responsible for handling events and drawing components on the screen. This GUI thread is created when the first window is opened. So in fact, you have already done parallel programming! When a `main` routine opens a window, both the main thread and the GUI thread can continue to run in parallel. Of course, parallel programming can be used in much more interesting ways.

Unfortunately, parallel programming is even more difficult than ordinary, single-threaded programming. When several threads are working together on a problem, a whole new category of errors is possible. This just means that techniques for writing correct and robust programs are even more important for parallel programming than they are for normal programming. On the other hand, fortunately, Java has a nice thread API that makes basic uses of threads reasonably easy. It also has some standard classes to help with some of the more tricky parts. It won't be until midway through Section 12.3 that you'll learn about the low-level techniques that are necessary to handle the trickiest parts of parallel programming.

12.1.1 Creating and Running Threads

In Java, a thread is represented by an object belonging to the class `java.lang.Thread` (or to a subclass of this class). The purpose of a *Thread* object is to execute a single method and to execute it just once. This method represents the task to be carried out by the thread. The method is executed in its own thread of control, which can run in parallel with other threads. When the execution of the thread's method is finished, either because the method terminates normally or because of an uncaught exception, the thread stops running. Once this happens, there is no way to restart the thread or to use the same *Thread* object to start another thread.

There are two ways to program a thread. One is to create a subclass of *Thread* and to define the method `public void run()` in the subclass. This `run()` method defines the task that will be performed by the thread; that is, when the thread is started, it is the `run()` method that will be executed in the thread. For example, here is a simple, and rather useless, class that defines a thread that does nothing but print a message on standard output:

```
public class NamedThread extends Thread {
   private String name;  // The name of this thread.
   public NamedThread(String name) {  // Constructor gives name to thread.
      this.name = name;
   }
   public void run() {  // The run method prints a message to standard output.
      System.out.println("Greetings from thread '" + name + "'!");
   }
}
```

To use a *NamedThread*, you must of course create an object belonging to this class. For example,

```
NamedThread greetings = new NamedThread("Fred");
```

However, creating the object does not automatically start the thread running or cause its `run()` method to be executed. To do that, you must call the `start()` method in the thread object. For the example, this would be done with the statement

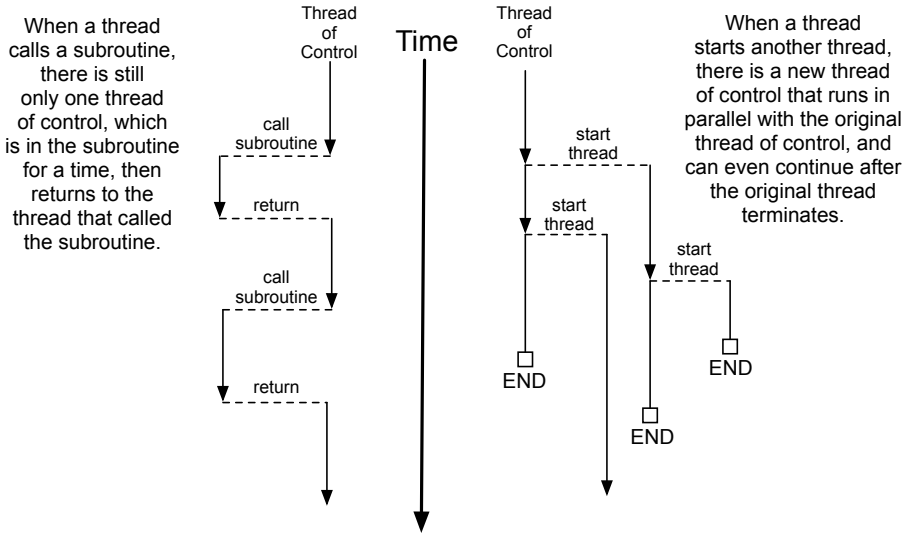
```
greetings.start();
```

The purpose of the `start()` method is to create the new thread of control that will execute the *Thread* object's `run()` method. The new thread runs in parallel with the thread in which the `start()` method was called, along with any other threads that already existed. The `start()` method returns immediately after starting the new thread of control, **without waiting for the thread to terminate**. This means that the code in the thread's `run()` method executes at the same time as the statements that follow the call to the `start()` method. Consider this code segment:

```
NamedThread greetings = new NamedThread("Fred");
greetings.start();
System.out.println("Thread has been started");
```

After `greetings.start()` is executed, there are two threads. One of them will print "Thread has been started" while the other one wants to print "Greetings from thread 'Fred'!". It is important to note that *these messages can be printed in either order*. The two threads run simultaneously and will compete for access to standard output, so that they can print their messages. Whichever thread happens to be the first to get access will be the first to print its message. In a normal, single-threaded program, things happen in a definite, predictable order from beginning to end. In a multi-threaded program, there is a fundamental indeterminacy. You can't be sure what order things will happen in. This indeterminacy is what makes parallel programming so difficult!

Note that calling `greetings.start()` is **very** different from calling `greetings.run()`. Calling `greetings.run()` would execute the `run()` method in the same thread, rather than creating a new thread. This means that all the work of the `run()` will be done before the computer moves on to the statements that follow the call to `greetings.run()`. There is no parallelism and no indeterminacy.



This discussion has assumed that the computer on which the program is running has more than one processing unit, so that it is possible for the original thread and the newly created thread to literally be executed at the same time. However, it's possible to create multiple threads even on a computer that has only one processor (and, more generally, it is possible to create many more threads than there are processors, on any computer). In that case, the two threads will compete for time on the processor. However, there is still indeterminacy because the processor can switch from one thread to another at unpredictable times. In fact, from the point of view of the programmer, there is no difference between programming for a single-processor computer and programming for a multi-processor computer, and we will pretty much ignore the distinction from now on.

* * *

I mentioned that there are two ways to program a thread. The first way was to define a subclass of *Thread*. The second is to define a class that implements the interface `java.lang.Runnable`. The *Runnable* interface defines a single method, `public void run()`. Given a *Runnable*, it is possible to create a *Thread* whose task is to execute the *Runnable's* `run()` method.

The *Thread* class has a constructor that takes a *Runnable* as its parameter. When an object that implements the *Runnable* interface is passed to that constructor, the `run()` method of the thread will simply call the `run()` method from the *Runnable*, and calling the thread's `start()` method will create a new thread of control in which the *Runnable's* `run()` method is executed. For example, as an alternative to the *NamedThread* class, we could define the class:

```
public class NamedRunnable implements Runnable {
   private String name;  // The name of this Runnable.
   public NamedRunnable(String name) {  // Constructor gives name to object.
      this.name = name;
   }
   public void run() {  // The run method prints a message to standard output.
      System.out.println("Greetings from runnable '" + name +"'!");
   }
}
```

To use this version of the class, we would create a *NamedRunnable* object and use that object to create an object of type *Thread*:

```
NamedRunnable greetings = new NamedRunnable("Fred");
Thread greetingsThread = new Thread(greetings);
greetingsThread.start();
```

The advantage of doing things this way is that **any** object can implement the *Runnable* interface and can contain a `run()` method, which can then be executed in a separate thread. That `run()` method has access to everything in the class, including `private` variables and methods. The disadvantage is that this way of doing things is not very object-oriented: It violates the principle that each object should have a single, clearly-defined responsibility. Instead of making some random object *Runnable* just so that you can use it to make a thread, you can consider using a nested inner subclass of the *Thread* class to define the thread. (See Subsection 5.7.2.)

Finally, I'll note that it is sometimes convenient to define a thread using an **anonymous** inner class (Subsection 5.7.3). For example:

```
Thread greetingsFromFred = new Thread() {
   public void run() {
      System.out.println("Greetings from Fred!");
   }
};
greetingsFromFred.start();
```

* * *

To help you understand how multiple threads are executed in parallel, we consider the sample program *ThreadTest1.java*. This program creates several threads. Each thread performs exactly the same task. The task is to count the number of integers less than 1000000 that are prime. (The particular task that is done is not important for our purposes here.) This computation should take less than a second on a modern computer. The threads that perform this task are defined by the following static nested class:

```
/**
 * When a thread belonging to this class is run it will count the
 * number of primes between 2 and 1000000.  It will print the result
 * to standard output, along with its ID number and the elapsed
 * time between the start and the end of the computation.
 */
private static class CountPrimesThread extends Thread {
   int id;  // An id number for this thread; specified in the constructor.
   public CountPrimesThread(int id) {
      this.id = id;
   }
   public void run() {
      long startTime = System.currentTimeMillis();
      int count = countPrimes(2,1000000); // Counts the primes.
      long elapsedTime = System.currentTimeMillis() - startTime;
      System.out.println("Thread " + id + " counted " +
            count + " primes in " + (elapsedTime/1000.0) + " seconds.");
   }
}
```

The main program asks the user how many threads to run, and then creates and starts the specified number of threads:

```
public static void main(String[] args) {
   int numberOfThreads = 0;
   while (numberOfThreads < 1 || numberOfThreads > 25) {
      System.out.print("How many threads do you want to use  (1 to 25) ?  ");
      numberOfThreads = TextIO.getlnInt();
      if (numberOfThreads < 1 || numberOfThreads > 25)
         System.out.println("Please enter a number between 1 and 25 !");
   }
   System.out.println("\nCreating " + numberOfThreads
                                          + " prime-counting threads...");
   CountPrimesThread[] worker = new CountPrimesThread[numberOfThreads];
   for (int i = 0; i < numberOfThreads; i++)
      worker[i] = new CountPrimesThread( i );
   for (int i = 0; i < numberOfThreads; i++)
      worker[i].start();
   System.out.println("Threads have been created and started.");
}
```

It would be a good idea for you to compile and run the program or to try the applet version, which can be found in the on-line version of this section.

When I ran the program with one thread on a rather old laptop, it took 1.18 seconds for the computer to do the computation. When I ran it using six threads, the output was:

```
Creating 6 prime counting threads...
Threads have been created and started.
Thread 1 counted 78498 primes in 6.706 seconds.
Thread 4 counted 78498 primes in 6.693 seconds.
Thread 0 counted 78498 primes in 6.838 seconds.
Thread 2 counted 78498 primes in 6.825 seconds.
Thread 3 counted 78498 primes in 6.893 seconds.
Thread 5 counted 78498 primes in 6.859 seconds.
```

The second line was printed immediately after the first. At this point, the main program has ended but the six threads continue to run. After a pause of about seven seconds, all six threads completed at about the same time. The order in which the threads complete is not the same as the order in which they were started, and the order is indeterminate. That is, if the program is run again, the order in which the threads complete will probably be different.

On this computer, six threads took about six times longer than one thread. This is because the computer had only one processor. Six threads, all doing the same task, take six times as much processing as one thread. With only one processor to do the work, the total elapsed time for six threads is about six times longer than the time for one thread. On a computer with two processors, the computer can work on two tasks at the same time, and six threads might complete in as little as three times the time it takes for one thread. On a computer with six or more processors, six threads might take no more time than a single thread. Because of overhead and other reasons, the actual speedup will probably be a little smaller than this analysis indicates, but on a multiprocessor machine, you should see a definite speedup. What happens when you run the program on your own computer? How many processors do you have?

Whenever there are more threads to be run than there are processors to run them, the computer divides its attention among all the runnable threads by switching rapidly from one thread to another. That is, each processor runs one thread for a while then switches to another thread and runs that one for a while, and so on. Typically, these "context switches" occur about 100 times or more per second. The result is that the computer makes progress on all

the tasks, and it looks to the user as if all the tasks are being executed simultaneously. This is why in the sample program, in which each thread has the same amount of work to do, all the threads complete at about the same time: Over any time period longer than a fraction of a second, the computer's time is divided approximately equally among all the threads.

12.1.2 Operations on Threads

Much of Java's thread API can be found in the *Thread* class. However, we'll start with a thread-related method in *Runtime*, a class that allows a Java program to get information about the environment in which it is running. When you do parallel programming in order to spread the work among several processors, you might want to take into account the number of available processors. You might, for example, want to create one thread for each processor. In Java, you can find out the number of processors by calling the function

```
Runtime.getRuntime().availableProcessors()
```

which returns an **int** giving the number of processors that are available to the Java Virtual Machine. In some cases, this might be less than the actual number of processors in the computer.

* * *

A *Thread* object contains several useful methods for working with threads. Most important is the `start()` method, which was discussed above.

Once a thread has been started, it will continue to run until its `run()` method ends for some reason. Sometimes, it's useful for one thread to be able to tell whether another thread has terminated. If `thrd` is an object of type *Thread*, then the **boolean**-valued function `thrd.isAlive()` can be used to test whether or not `thrd` has terminated. A thread is "alive" between the time it is started and the time when it terminates. After the thread has terminated it is said to be "dead." (The rather gruesome metaphor is also used when we refer to "killing" or "aborting" a thread.) Remember that a thread that has terminated cannot be restarted.

The static method `Thread.sleep(milliseconds)` causes the thread that executes this method to "sleep" for the specified number of milliseconds. A sleeping thread is still alive, but it is not running. While a thread is sleeping, the computer can work on any other runnable threads (or on other programs). `Thread.sleep()` can be used to insert a pause in the execution of a thread. The `sleep()` method can throw an exception of type *InterruptedException*, which is a checked exception that requires mandatory exception handling. In practice, this means that the `sleep()` method is usually called inside a `try..catch` statement that catches the potential *InterruptedException*:

```
try {
   Thread.sleep(lengthOfPause);
}
catch (InterruptedException e) {
}
```

One thread can ***interrupt*** another thread to wake it up when it is sleeping or paused for certain other reasons. A *Thread*, `thrd`, can be interrupted by calling the method `thrd.interrupt()`. Doing so can be a convenient way to send a signal from one thread to another. A thread knows it has been interrupted when it catches an *InterruptedException*. Outside the `catch` handler for the exception, the thread can check whether it has been interrupted by calling the static method `Thread.interrupted()`. This method tells whether the current thread—the thread that executes the method—has been interrupted. It also has the unusual property of clearing

the interrupted status of the thread, so you only get one chance to check for an interruption. In your own programs, your threads are not going to be interrupted unless **you** interrupt them. So most often, you are not likely to need to do anything in response to an *InterruptedException* (except to `catch` it).

Sometimes, it's necessary for one thread to wait for another thread to die. This is done with the `join()` method from the *Thread* class. Suppose that `thrd` is a *Thread*. Then, if another thread calls `thrd.join()`, that other thread will go to sleep until `thrd` terminates. If `thrd` is already dead when `thrd.join()` is called, then it simply has no effect. The `join()` method can throw an *InterruptedException*, which must be handled as usual. As an example, the following code starts several threads, waits for them all to terminate, and then outputs the elapsed time:

```
CountPrimesThread[] worker = new CountPrimesThread[numberOfThreads];
long startTime = System.currentTimeMillis();
for (int i = 0; i < numberOfThreads; i++) {
   worker[i] = new CountPrimesThread();
   worker[i].start();
}
for (int i = 0; i < numberOfThreads; i++) {
   try {
      worker[i].join();  // Wait until worker[i] finishes, if it hasn't already.
   }
   catch (InterruptedException e) {
   }
}
// At this point, all the worker threads have terminated.
long elapsedTime = System.currentTimeMillis() - startTime;
System.out.println("Total elapsed time: " + (elapsedTime/1000.0) + " seconds");
```

An observant reader will note that this code assumes that no *InterruptedException* will occur. To be absolutely sure that the thread `worker[i]` has terminated in an environment where *InterruptedExceptions* are possible, you would have to do something like:

```
while (worker[i].isAlive()) {
   try {
      worker[i].join();
   }
   catch (InterruptedException e) {
   }
}
```

Another version of the `join()` method takes an integer parameter that specifies the maximum number of milliseconds to wait. A call to `thrd.join(m)` will wait until either `thrd` has terminated or until `m` milliseconds have elapsed. This can be used to allow a thread to wake up occasionally to perform some task while it is waiting. Here, for example, is a code segment that will start a thread, `thrd`, and then will output a period every two seconds as long as `thrd` continues to run:

```
System.out.print("Running the thread ");
thrd.start();
while (thrd.isAlive()) {
   try {
      thrd.join(2000);
      System.out.print(".");
   }
```

```
        catch (InterruptedException e) {
        }
    }
    System.out.println(" Done!");
```

* * *

Threads have two properties that are occasionally useful: a daemon status and a priority. A *Thread* `thrd` can be designated as a ***daemon thread*** by calling `thrd.setDaemon(true)`. This must be done before the thread is started, and it can throw an exception of type *SecurityException* if the calling thread is not allowed to modify `thrd's` properties. This has only one effect: The Java Virtual Machine will exit as soon as there are no **non-daemon** threads that are still alive. That is, the fact that a daemon thread is still alive is not enough to keep the Java Virtual Machine running. A daemon thread might exist, for example, only to provide some service to other, non-daemon threads. When there are no more non-daemon threads, there will be no further call for the daemon thread's services, so the program might as well shut down.

The priority of a thread is a more important property. Every thread has a ***priority***, specified as an integer. A thread with a greater priority value will be run in preference to a thread with a smaller priority. For example, computations that can be done in the background, when no more important thread has work to do, can be run with a low priority. In the next section, we will see how this can be useful in GUI programs. If `thrd` is of type *Thread*, then `thrd.getPriority()` returns the integer that specifies `thrd`'s priority, and `thrd.setPriority(p)` can be used to set its priority to a given integer, `p`.

Priorities cannot be arbitrary integers, and `thrd.setPriority()` will throw an *IllegalArgumentException* if the specified priority is not in the legal range for the thread. The range of legal priority values can differ from one computer to another. The range of legal values is specified by the constants `Thread.MIN_PRIORITY` and `Thread.MAX_PRIORITY`, but a given thread might be further restricted to values less than `Thread.MAX_PRIORITY`. The default priority is given by `Thread.NORM_PRIORITY`. To set `thrd` to run with a priority value just below the normal priority, you can call

```
thrd.setPriority( Thread.NORM_PRIORITY - 1 );
```

Note that `thrd.setPriority()` can also throw an exception of type *SecurityException*, if the thread that calls the method is not allowed to set the priority of `thrd`.

Finally, I'll note that the static method `Thread.currentThread()` returns the current thread. That is, the return value of this method is the thread that executed the method. This allows a thread to get a reference to itself, so that it can modify its own properties. For example, you can determine the priority of the currently running thread by calling `Thread.currentThread().getPriority()`.

12.1.3 Mutual Exclusion with "synchronized"

It's pretty easy to program several threads to carry out completely independent tasks. The real difficulty arises when threads have to interact in some way. One way that threads interact is by sharing resources. When two threads need access to the same resource, such as a variable or a window on the screen, some care must be taken that they don't try to use the same resource at the same time. Otherwise, the situation could be something like this: Imagine several cooks sharing the use of just one measuring cup, and imagine that Cook A fills the measuring cup with milk, only to have Cook B grab the cup before Cook A has a chance to empty the milk

into his bowl. There has to be some way for Cook A to claim exclusive rights to the cup while he performs the two operations: Add-Milk-To-Cup and Empty-Cup-Into-Bowl.

Something similar happens with threads, even with something as simple as adding one to a counter. The statement

```
count = count + 1;
```

is actually a sequence of three operations:

```
Step 1.  Get the value of count
Step 2.  Add 1 to the value.
Step 3.  Store the new value in count
```

Suppose that several threads perform these three steps. Remember that it's possible for two threads to run at the same time, and even if there is only one processor, it's possible for that processor to switch from one thread to another at any point. Suppose that while one thread is between Step 2 and Step 3, another thread starts executing the same sequence of steps. Since the first thread has not yet stored the new value in `count`, the second thread reads the **old** value of `count` and adds one to that old value. Both threads have computed the same new value for `count`, and both threads then go on to store that value back into `count` by executing Step 3. After both threads have done so, the value of `count` has gone up only by 1 instead of by 2! This type of problem is called a ***race condition***. This occurs when one thread is in the middle of a multi-step operation, and another thread can change some value or condition that the first thread is depending upon. (The first thread is "in a race" to complete all the steps before it is interrupted by another thread.)

Another example of a race condition can occur in an `if` statement. Consider the following statement, which is meant to avoid a division-by-zero error:

```
if ( A != 0 )
   B = C / A;
```

Suppose that this statement is executed by some thread. If the variable `A` is shared by one or more other threads, and if nothing is done to guard against the race condition, then it is possible that one of those other threads will change the value of `A` to zero between the time that the first thread checks the condition `A != 0` and the time that it does the division. This means that the thread can end up dividing by zero, even though it just checked that `A` was not zero!

To fix the problem of race conditions, there has to be some way for a thread to get ***exclusive access*** to a shared resource. This is not a trivial thing to implement, but Java provides a high-level and relatively easy-to-use approach to exclusive access. It's done with ***synchronized methods*** and with the ***synchronized statement***. These are used to protect shared resources by making sure that only one thread at a time will try to access the resource. Synchronization in Java actually provides only ***mutual exclusion***, which means that exclusive access to a resource is only guaranteed if **every** thread that needs access to that resource uses synchronization. Synchronization is like a cook leaving a note that says, "I'm using the measuring cup." This will get the cook exclusive access to the cup—but only if all the cooks agree to check the note before trying to grab the cup.

Because this is a difficult topic, I will start with a simple example. Suppose that we want to avoid the race condition that occurs when several threads all want to add 1 to a counter. We can do this by defining a class to represent the counter and by using synchronized methods in that class. A method is declared to be synchronized by adding the reserved word `synchronized` as a modifier to the definition of the method:

```
public class ThreadSafeCounter {

   private int count = 0;  // The value of the counter.

   synchronized public void increment() {
      count = count + 1;
   }

   synchronized public int getValue() {
      return count;
   }

}
```

If `tsc` is of type *ThreadSafeCounter*, then any thread can call `tsc.increment()` to add 1 to the counter in a completely safe way. The fact that `tsc.increment()` is `synchronized` means that only one thread can be in this method at a time; once a thread starts executing this method, it is guaranteed that it will finish executing it without having another thread change the value of `tsc.count` in the meantime. There is no possibility of a race condition. Note that the guarantee depends on the fact that `count` is a `private` variable. This forces all access to `tsc.count` to occur in the `synchronized` methods that are provided by the class. If `count` were `public`, it would be possible for a thread to bypass the synchronization by, for example, saying `tsc.count++`. This could change the value of `count` while another thread is in the middle of `tsc.increment()`. Remember that synchronization by itself does **not** guarantee exclusive access; it only guarantees **mutual exclusion** among all the threads that are synchronized.

The *ThreadSafeCounter* class does not prevent all possible race conditions that might arise when using a counter. Consider the `if` statement:

```
if ( tsc.getValue() == 0 )
   doSomething();
```

where `doSomething()` is some method that requires the value of the counter to be zero. There is still a race condition here, which occurs if a second thread increments the counter between the time the first thread tests `tsc.getValue() == 0` and the time it executes `doSomething()`. The first thread needs exclusive access to the counter during the execution of the whole `if` statement. (The synchronization in the *ThreadSafeCounter* class only gives it exclusive access during the time it is evaluating `tsc.getValue()`.) We can solve the race condition by putting the `if` statement in a `synchronized` statement:

```
synchronized(tsc) {
   if ( tsc.getValue() == 0 )
      doSomething();
}
```

Note that the `synchronized` statement takes an object—`tsc` in this case—as a kind of parameter. The syntax of the `synchronized` statement is:

```
synchronized( ⟨object⟩ ) {
   ⟨statements⟩
}
```

In Java, mutual exclusion is always associated with an object; we say that the synchronization is "on" that object. For example, the `if` statement above is "synchronized on `tsc`." A synchronized instance method, such as those in the class *ThreadSafeCounter*, is synchronized on the object that contains the instance method. In fact, adding the `synchronized` modifier to the

definition of an instance method is pretty much equivalent to putting the body of the method in a `synchronized` statement of the form `synchronized(this) {...}`. It is also possible to have synchronized static methods; a synchronized static method is synchronized on the special class object that represents the class containing the static method.

The real rule of synchronization in Java is this: **Two threads cannot be synchronized on the same object at the same time**; that is, they cannot simultaneously be executing code segments that are synchronized on that object. If one thread is synchronized on an object, and a second thread tries to synchronize on the **same** object, the second thread is forced to wait until the first thread has finished with the object. This is implemented using something called a ***synchronization lock***. Every object has a synchronization lock, and that lock can be "held" by only one thread at a time. To enter a synchronized statement or synchronized method, a thread must obtain the associated object's lock. If the lock is available, then the thread obtains the lock and immediately begins executing the synchronized code. It releases the lock after it finishes executing the synchronized code. If Thread A tries to obtain a lock that is already held by Thread B, then Thread A has to wait until Thread B releases the lock. In fact, Thread A will go to sleep, and will not be awoken until the lock becomes available.

* * *

As a simple example of shared resources, we return to the prime-counting problem. In this case, instead of having every thread perform exactly the same task, we'll do some real parallel processing. The program will count the prime numbers in a given range of integers, and it will do so by dividing the work up among several threads. Each thread will be assigned a part of the full range of integers, and it will count the primes in its assigned part. At the end of its computation, the thread has to add its count to the overall total of primes in the entire range. The variable that represents the total is shared by all the threads, since each thread has to add a number to the total. If each thread just says

```
total = total + count;
```

then there is a (small) chance that two threads will try to do this at the same time and that the final total will be wrong. To prevent this race condition, access to `total` has to be synchronized. My program uses a synchronized method to add the counts to the total. This method is called once by each thread:

```
synchronized private static void addToTotal(int x) {
   total = total + x;
   System.out.println(total + " primes found so far.");
}
```

The source code for the program can be found in *ThreadTest2.java*. This program counts the primes in the range 3000001 to 6000000. (The numbers are rather arbitrary.) The `main()` routine in this program creates between 1 and 5 threads and assigns part of the job to each thread. It waits for all the threads to finish, using the `join()` method as described above. It then reports the total number of primes found, along with the elapsed time. Note that `join()` is required here, since it doesn't make sense to report the number of primes until all of the threads have finished.

If you run the program on a multiprocessor computer, it should take less time for the program to run when you use more than one thread. You can compile and run the program or try the equivalent applet in the on-line version of this section.

* * *

Synchronization can help to prevent race conditions, but it introduces the possibility of another type of error, ***deadlock***. A deadlock occurs when a thread waits forever for a resource that it will never get. In the kitchen, a deadlock might occur if two very simple-minded cooks both want to measure a cup of milk at the same time. The first cook grabs the measuring cup, while the second cook grabs the milk. The first cook needs the milk, but can't find it because the second cook has it. The second cook needs the measuring cup, but can't find it because the first cook has it. Neither cook can continue and nothing more gets done. This is deadlock. Exactly the same thing can happen in a program, for example if there are two threads (like the two cooks) both of which need to obtain locks on the same two objects (like the milk and the measuring cup) before they can proceed. Deadlocks can easily occur, unless great care is taken to avoid them.

12.1.4 Volatile Variables

Synchronization is only one way of controlling communication among threads. We will cover several other techniques later in the chapter. For now, we finish this section with one more communication technique: volatile variables.

In general, threads communicate by sharing variables and accessing those variables in synchronized methods or synchronized statements. However, synchronization is fairly expensive computationally, and excessive use of it should be avoided. So in some cases, it can make sense for threads to refer to shared variables without synchronizing their access to those variables.

However, a subtle problem arises when the value of a shared variable is set in one thread and used in another. Because of the way that threads are implemented in Java, the second thread might not see the changed value of the variable immediately. That is, it is possible that a thread will continue to see the **old** value of the shared variable for some time after the value of the variable has been changed by another thread. This is because threads are allowed to ***cache*** shared data. That is, each thread can keep its own local copy of the shared data. When one thread changes the value of a shared variable, the local copies in the caches of other threads are not immediately changed, so the other threads can continue to see the old value, at least briefly.

When a `synchronized` method or statement is entered, threads are forced to update their caches to the most current values of the variables in the cache. So, using shared variables in synchronized code is always safe.

It is possible to use a shared variable safely **outside** of synchronized code, but in that case, the variable must be declared to be ***volatile***. The `volatile` keyword is a modifier that can be added to a variable declaration, as in

```
private volatile int count;
```

If a variable is declared to be `volatile`, no thread will keep a local copy of that variable in its cache. Instead, the thread will always use the official, main copy of the variable. This means that any change that is made to the variable will immediately be visible to all threads. This makes it safe for threads to refer to `volatile` shared variables even outside of synchronized code. Access to volatile variables is less efficient than access to non-volatile variables, but more efficient than using synchronization. (Remember, though, that synchronization is still the only way to prevent race conditions.)

When the `volatile` modifier is applied to an object variable, only the variable itself is declared to be volatile, not the contents of the object that the variable points to. For this

reason, `volatile` is used mostly for variables of simple types such as primitive types and enumerated types.

A typical example of using volatile variables is to send a signal from one thread to another that tells the second thread to terminate. The two threads would share a variable

```
volatile boolean terminate = false;
```

The run method of the second thread would check the value of `terminate` frequently, and it would end when the value of `terminate` becomes true:

```
public void run() {
   while ( terminate == false ) {
      .
      .  // Do some work.
      .
   }
}
```

This thread will run until some other thread sets the value of `terminate` to true. Something like this is really the only clean way for one thread to cause another thread to die.

(By the way, you might be wondering why threads should use local data caches in the first place, since it seems to complicate things unnecessarily. Caching is allowed because of the structure of multiprocessing computers. In many multiprocessing computers, each processor has some local memory that is directly connected to the processor. A thread's cache can be stored in the local memory of the processor on which the thread is running. Access to this local memory is much faster than access to other memory, so it is more efficient for a thread to use a local copy of a shared variable rather than some "master copy" that is stored in non-local memory.)

12.2 Programming with Threads

Threads introduce new complexity into programming, but they are an important tool and will only become more essential in the future. So, every programmer should know some of the fundamental design patterns that are used with threads. In this section, we will look at some basic techniques, with more to come as the chapter progresses.

12.2.1 Threads Versus Timers

One of the most basic uses of threads is to perform some period task at set intervals. In fact, this is so basic that there is a specialized class for performing this task—and you've already worked with it. The *Timer* class, in package `javax.swing`, can generate a sequence of *ActionEvents* separated by a specified time interval. Timers were introduced in Section 6.5, where they were used to implement animations. Behind the scenes, a *Timer* uses a thread. The thread sleeps most of the time, but it wakes up periodically to generate the events associated with the timer. Before timers were introduced, threads had to be used directly to implement a similar functionality.

In a typical use of a timer for animation, each event from the timer causes a new frame of the animation to be computed and displayed. In the response to the event, it is only necessary to update some state variables and to repaint the display to reflect the changes. A *Timer* to do that every thirty milliseconds might be created like this:

```
Timer timer = new Timer( 30, new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        updateForNextFrame();
        display.repaint();
    }
} );
timer.start();
```

Suppose that we wanted to do the same thing with a thread. The `run()` method of the thread would have to execute a loop in which the thread sleeps for 30 milliseconds, then wakes up to do the updating and repainting. This could be implemented in a nested class as follows using the method `Thread.sleep()` that was discussed in Subsection 12.1.2:

```
private class Animator extends Thread {
    public void run() {
       while (true) {
           try {
               Thread.sleep(30);
           }
           catch (InterruptedException e) {
           }
           updateForNextFrame();
           display.repaint();
       }
    }
}
```

To run the animation, you would create an object belonging to this class and call its `start()` method. As it stands, there would be no way to stop the animation once it is started. One way to make it possible to stop the animation would be to end the loop when a `volatile` **boolean** variable, `terminate`, becomes `true`, as discussed in Subsection 12.1.4. Since thread objects can only be executed once, in order to restart the animation after it has been stopped, it would be necessary to create a new thread. In the next section, we'll see some more versatile techniques for controlling threads.

There is a subtle difference between using threads and using timers for animation. The thread that is used by a Swing *Timer* does nothing but generate events. The event-handling code that is executed in response to those events is actually executed in the Swing event-handling thread, which also handles repainting of components and responses to user actions. This is important because the Swing GUI is not ***thread-safe***. That is, it does not use synchronization to avoid race conditions among threads trying to access GUI components and their state variables. As long as everything is done in the Swing event thread, there is no problem. A problem can arise when another thread manipulates components or the variables that they use. In the *Animator* example given above, this could happen when the thread calls the `updateForNextFrame()` method. The variables that are modified in `updateForNextFrame()` would also be used by the `paintComponent()` method that draws the frame. There is a race condition here: If these two methods are being executed simultaneously, there is a possibility that `paintComponent()` will use inconsistent variable values—some appropriate for the new frame, some for the previous frame.

One solution to this problem would be to declare both `paintComponent()` and `updateForNextFrame()` to be `synchronized` methods. The real solution in this case is to use a timer rather than a thread. In practice, race conditions are not likely to be an issue for

simple animations, even if they are implemented using threads. But it can become a real issue when threads are used for more complex tasks.

I should note that the `repaint()` method of a component can be safely called from any thread, without worrying about synchronization. Recall that `repaint()` does not actually do any painting itself. It just tells the system to schedule a paint event. The actual painting will be done later, in the Swing event-handling thread. I will also note that Java has another timer class, *java.util.Timer*, that is appropriate for use in non-GUI programs.

The sample program *RandomArtWithThreads.java* uses a thread to drive a very simple animation. You can compare it to *RandomArtPanel.java*, from Section 6.5, which implemented the same animation with a timer.

12.2.2 Recursion in a Thread

Although timers should be used in preference to threads when possible, there are times when it is reasonable to use a thread even for a straightforward animation. One reason to do so is when the thread is running a recursive algorithm, and you want to repaint the display many times over the course of the recursion. (Recursion is covered in Section 9.1.) It's difficult to drive a recursive algorithm with a series of events from a timer; it's much more natural to use a single recursive method call to do the recursion, and it's easy to do that in a thread.

As an example, the program *QuicksortThreadDemo.java* uses an animation to illustrate the recursive QuickSort algorithm for sorting an array. In this case, the array contains colors, and the goal is to sort the colors into a standard spectrum from red to violet. You can see the program as an applet in the on-line version of this section. In the program, the user randomizes the array and starts the sorting process by clicking the "Start" button below the display. The "Start" button changes to a "Finish" button that can be used to abort the sort before it finishes on its own.

In this program, the display's `repaint()` method is called every time the algorithm makes a change to the array. Whenever this is done, the thread sleeps for 100 milliseconds to allow time for the display to be repainted and for the user to see the change. There is also a longer delay, one full second, just after the array is randomized, before the sorting starts. Since these delays occur at several points in the code, `QuicksortThreadDemo` defines a `delay()` method that makes the thread that calls it sleep for a specified period. The `delay()` method calls `display.repaint()` just before sleeping. While the animation thread sleeps, the event-handling thread will have a chance to run and will have plenty of time to repaint the display.

An interesting question is how to implement the "Finish" button, which should abort the sort and terminate the thread. Pressing this button causes the value of a `volatile` **boolean** variable, `running`, to be set to `false`, as a signal to the thread that it should terminate. The problem is that this button can be clicked at any time, even when the algorithm is many levels down in the recursion. Before the thread can terminate, all of those recursive method calls must return. A nice way to cause that is to throw an exception. `QuickSortThreadDemo` defines a new exception class, *ThreadTerminationException*, for this purpose. The `delay()` method checks the value of the signal variable, `running`. If `running` is `false`, the `delay()` method throws the exception that will cause the recursive algorithm, and eventually the animation thread itself, to terminate. Here, then, is the `delay()` method:

```
private void delay(int millis) {
   if (! running)
      throw new ThreadTerminationException();
   display.repaint();
```

```
      try {
         Thread.sleep(millis);
      }
      catch (InterruptedException e) {
      }
      if (! running) // Check again, in case it changed during the sleep period.
         throw new ThreadTerminationException();
   }
```

The *ThreadTerminationException* is caught in the thread's `run()` method:

```
   /**
    * This class defines the threads that run the recursive
    * QuickSort algorithm.  The thread begins by randomizing the
    * array, hue.  It then calls quickSort() to sort the entire array.
    * If quickSort() is aborted by a ThreadTerminationException,
    * which would be caused by the user clicking the Finish button,
    * then the thread will restore the array to sorted order before
    * terminating, so that whether or not the quickSort is aborted,
    * the array ends up sorted.
    */
   private class Runner extends Thread {
      public void run() {
         try {
            for (int i = hue.length-1; i > 0; i--) { // Randomize array.
               int r = (int)((i+1)*Math.random());
               int temp = hue[r];
               hue[r] = hue[i];
               hue[i] = temp;
            }
            delay(1000);  // Wait one second before starting the sort.
            quickSort(0,hue.length-1);  // Sort the whole array, recursively.
         }
         catch (ThreadTerminationException e) { // User clicked "Finish".
            for (int i = 0; i < hue.length; i++)
               hue[i] = i;
         }
         finally {// Make sure running is false and button label is correct.
            running = false;
            startButton.setText("Start");
            display.repaint();
         }
      }
   }
```

The program uses a variable, `runner`, of type *Runner* to represent the thread that does the sorting. When the user clicks the "Start" button, the following code is executed to create and start the thread:

```
   startButton.setText("Finish");
   runner = new Runner();
   running = true;  // Set the signal before starting the thread!
   runner.start();
```

Note that the value of the signal variable `running` is set to `true` before starting the thread. If `running` were `false` when the thread was started, the thread might see that value as soon as it starts and interpret it as a signal to stop before doing anything. Remember that when `runner.start()` is called, `runner` starts running in parallel with the thread that called it.

Stopping the thread is a little more interesting, because the thread might be sleeping when the "Finish" button is pressed. The thread has to wake up before it can act on the signal that it is to terminate. To make the thread a little more responsive, we can call `runner.interrupt()`, which will wake the thread if it is sleeping. (See Subsection 12.1.2.) This doesn't have much practical effect in this program, but it does make the program respond noticeably more quickly if the user presses "Finish" immediately after pressing "Start," while the thread is sleeping for a full second.

12.2.3 Threads for Background Computation

In order for a GUI program to be responsive—that is, to respond to events very soon after they are generated—it's important that event-handling methods in the program finish their work very quickly. Remember that events go into a queue as they are generated, and the computer cannot respond to an event until after the event-handler methods for previous events have done their work. This means that while one event handler is being executed, other events will have to wait. If an event handler takes a while to run, the user interface will effectively freeze up during that time. This can be very annoying if the delay is more than a fraction of a second. Fortunately, modern computers can do an awful lot of computation in a fraction of a second.

However, some computations are too big to be done in event handlers. The solution, in that case, is to do the computation in another thread that runs in parallel with the event-handling thread. This makes it possible for the computer to respond to user events even while the computation is ongoing. We say that the computation is done "in the background."

Note that this application of threads is very different from the previous example. When a thread is used to drive a simple animation, it actually does very little work. The thread only has to wake up several times each second, do a few computations to update state variables for the next frame of the animation, and call `repaint()` to cause the next frame to be displayed. There is plenty of time while the thread is sleeping for the computer to redraw the display and handle any other events generated by the user.

When a thread is used for background computation, however, we want to keep the computer as busy as possible working on the computation. The thread will compete for processor time with the event-handling thread; if you are not careful, event-handling—repainting in particular—can still be delayed. Fortunately, you can use thread priorities to avoid the problem. By setting the computation thread to run at a lower priority than the event-handling thread, you make sure that events will be processed as quickly as possible, while the computation thread will get all the extra processing time. Since event handling generally uses very little processing time, this means that most of the processing time goes to the background computation, but the interface is still very responsive. (Thread priorities were discussed in Subsection 12.1.2.)

The sample program *BackgroundComputationDemo.java* is an example of background processing. This program creates an image that takes some time to compute. The program uses some techniques for working with images that will not be covered until Subsection 13.1.1, for now all that you need to know is that it takes some computation to compute the color of each pixel in the image. The image itself is a piece of a mathematical object known as the Mandelbrot set. We will use the same image in several examples in this chapter, and will return to the Mandelbrot set in Section 13.5.

In outline, `BackgroundComputationDemo` is similar to the `QuicksortThreadDemo` discussed above. The computation is done in a thread defined by a nested class, *Runner*. A `volatile` **boolean** variable, `running`, is used to control the thread. If the value of `running` is set to `false`, the thread should terminate. The sample program has a button that the user clicks to start and to abort the computation. The difference is that the thread in this case is meant to run continuously, without sleeping. To allow the user to see that progress is being made in the computation (always a good idea), every time the thread computes a row of pixels, it copies those pixels to the image that is shown on the screen. The user sees the image being built up line-by-line.

When the computation thread is created in response to the "Start" button, we need to set it to run at a priority lower than the event-handling thread. The code that creates the thread is itself running in the event-handling thread, so we can use a priority that is one less than the priority of the thread that is executing the code. Note that the priority is set inside a `try..catch` statement. If an error occurs while trying to set the thread priority, the program will still work, though perhaps not as smoothly as it would if the priority was correctly set. Here is how the thread is created and started:

```
runner = new Runner();
try {
    runner.setPriority( Thread.currentThread().getPriority() - 1 );
}
catch (Exception e) {
    System.out.println("Error: Can't set thread priority: " + e);
}
running = true;  // Set the signal before starting the thread!
runner.start();
```

The other major point of interest in this program is that we have two threads that are both using the object that represents the image. The computation thread accesses the image in order to set the color of its pixels. The event-handling thread accesses the same image when it copies the image to the screen. Since the image is a resource that is shared by several threads, access to the image object should be synchronized. When the `paintComponent()` method copies the image to the screen (using a method that we have not yet covered), it does so in a `synchronized` statement:

```
synchronized(image) {
    g.drawImage(image,0,0,null);
}
```

When the computation thread sets the colors of a row of pixels (using another unfamiliar method), it also uses `synchronized`:

```
synchronized(image) {
    image.setRGB(0,row, width, 1, rgb, 0, width);
}
```

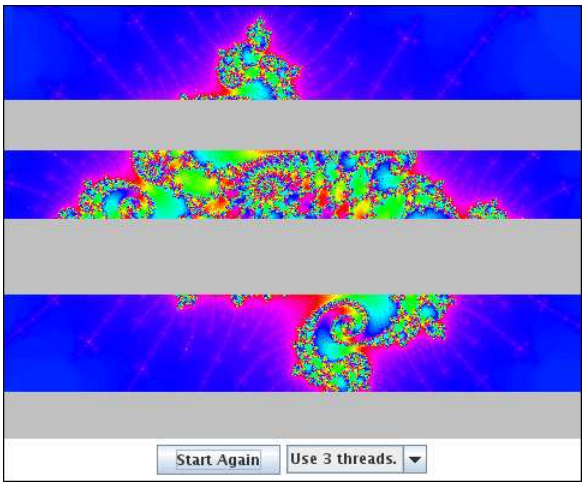
Note that both of these statements are synchronized on the same object, `image`. This is essential. In order to prevent the two code segments from being executed simultaneously, the synchronization must be on the same object. I use the `image` object here because it is convenient, but just about any object would do; it is **not** required that you synchronize on the object to which you are trying to control access.

Although `BackgroundComputationDemo` works OK, there is one problem: The goal is to get the computation done as quickly as possible, using all available processing time. The program

accomplishes that goal on a computer that has only one processor. But on a computer that has several processors, we are still using only **one** of those processors for the computation. It would be nice to get all the processors working on the problem. To do that, we need real parallel processing, with several computation threads. We turn to that problem next.

12.2.4 Threads for Multiprocessing

Our next example, *MultiprocessingDemo1.java*, is a variation on `BackgroundComputationDemo`. Instead of doing the computation in a single thread, `MultiprocessingDemo1` can divide the problem among several threads. The user can select the number of threads to be used. Each thread is assigned one section of the image to compute. The threads perform their tasks in parallel. For example, if there are two threads, the first thread computes the top half of the image while the second thread computes the bottom half. Here is picture of the program in the middle of a computation using three threads. The gray areas represent parts of the image that have not yet been computed:



You should try out the program. An applet version is on-line. On a multi-processor computer, the computation will complete more quickly when using several threads than when using just one. Note that when using one thread, this program has the same behavior as the previous example program.

The approach used in this example for dividing up the problem among threads is not optimal. We will see in the next section how it can be improved. However, `MultiprocessingDemo1` makes a good first example of multiprocessing.

When the user clicks the "Start" button, the program has to create and start the specified number of threads, and it has to assign a segment of the image to each thread. Here is how this is done:

```
workers = new Runner[threadCount];  // Holds the computation threads.
int rowsPerThread;  // How many rows of pixels should each thread compute?
rowsPerThread  = height / threadCount;  // (height = vertical size of image)
running = true;  // Set the signal before starting the threads!
threadsCompleted = 0;  // Records how many of the threads have terminated.
for (int i = 0; i < threadCount; i++) {
    int startRow;  // first row computed by thread number i
    int endRow;    // last row computed by thread number i
```

```
            // Create and start a thread to compute the rows of the image from
            // startRow to endRow.  Note that we have to make sure that
            // the endRow for the last thread is the bottom row of the image.
        startRow = rowsPerThread*i;
        if (i == threadCount-1)
            endRow = height-1;
        else
            endRow = rowsPerThread*(i+1) - 1;
        workers[i] = new Runner(startRow, endRow);
        try {
            workers[i].setPriority( Thread.currentThread().getPriority() - 1 );
        }
        catch (Exception e) {
        }
        workers[i].start();
    }
```

Beyond creating more than one thread, very few changes are needed to get the benefits of multiprocessing. Just as in the previous example, each time a thread has computed the colors for a row of pixels, it copies that row into the image, and synchronization is used in exactly the same way to control access to the image.

One thing is new, however. When all the threads have finished running, the name of the button in the program changes from "Abort" to "Start Again", and the pop-up menu, which has been disabled while the threads were running, is re-enabled. The problem is, how to tell when all the threads have terminated? (You might think about why we can't use `join()` to wait for the threads to end, as was done in the example in Subsection 12.1.2; at least, we can't do that in the event-handling thread!) In this example, I use an instance variable, `threadsCompleted`, to keep track of how many threads have terminated so far. As each thread finishes, it calls a method that adds one to the value of this variable. (The method is called in the `finally` clause of a `try` statement to make absolutely sure that it is called.) When the number of threads that have finished is equal to the number of threads that were created, the method updates the state of the program appropriately. Here is the method:

```
synchronized private void threadFinished() {
    threadsCompleted++;
    if (threadsCompleted == workers.length) { // All threads have finished.
        startButton.setText("Start Again");
        startButton.setEnabled(true);
        running = false; // Make sure running is false after the threads end.
        workers = null;  // Discard the array that holds the threads.
        threadCountSelect.setEnabled(true); // Re-enable pop-up menu.
    }
}
```

Note that this method is `synchronized`. This is to avoid the race condition when `threadsCompleted` is incremented. Without the synchronization, it is possible that two threads might call the method at the same time. If the timing is just right, both threads could read the same value for `threadsCompleted` and get the same answer when they increment it. The net result will be that `threadsCompleted` goes up by one instead of by two. One thread is not properly counted, and `threadsCompleted` will never become equal to the number of threads created. The program would hang in a kind of deadlock. The problem would occur only very

rarely, since it depends on exact timing. But in a large program, problems of this sort can be both very serious and very hard to debug. Proper synchronization makes the error impossible.

12.3 Threads and Parallel Processing

THE EXAMPLE AT THE END of the previous section used parallel processing to execute pieces of a large task. On a computer that has several processors, this allows the computation to be completed more quickly. However, the way that the program divided up the computation into subtasks was not optimal. Nor was the way that the threads were managed. In this section, we will look at two more versions of that program. The first improves the way the problem is decomposed into subtasks. The second improves the way threads are used. Along the way, I'll introduce a couple of built-in classes that Java provides to support parallel processing. Later in the section, I will cover `wait()` and `notify()`, lower-level methods that can be used to control parallel processes more directly.

12.3.1 Problem Decomposition

The sample program *MultiprocessingDemo1.java* divides the task of computing an image into several subtasks and assigns each subtask to a thread. While this works OK, there is a problem: Some of the subtasks might take substantially longer than others. The program divides the image up into equal parts, but the fact is that some parts of the image require more computation than others. In fact, if you run the program with three threads, you'll notice that the middle piece takes longer to compute than the top or bottom piece. In general, when dividing a problem into subproblems, it is very hard to predict just how much time it will take to solve each subproblem. Let's say that one particular subproblem happens to take a lot longer than all the others. The thread that computes that subproblem will continue to run for a relatively long time after all the other threads have completed. During that time, only **one** of the computer's processors will be working; the rest will be idle.

As a simple example, suppose that your computer has two processors. You divide the problem into two subproblems and create a thread to run each subproblem Your hope is that by using both processors, you can get your answer in half the time that it would take when using one processor. But if one subproblem takes four times longer than the other to solve, then for most of the time, only one processor will be working. In this case, you will only have cut the time needed to get your answer by 20%.

Even if you manage to divide your problem into subproblems that require equal amounts of computation, you still can't depend on all the subproblems requiring equal amounts of time to solve. For example, some of the processors on your computer might be busy running other programs. Or perhaps some of the processors are simply slower than others. (This is not so likely when running your computation on a single computer, but when distributing computation across several networked computers, as we will do later in this chapter, differences in processor speed can be a major issue.)

The common technique for dealing with all this is to divide the problem into a fairly large number of subproblems—many more subproblems than there are processors. This means that each processor will have to solve several subproblems. Each time a processor completes one subtask, it is assigned another subtask to work on, until all the subtasks have been assigned. Of course, there will still be variation in the time that the various subtasks require. One processor might complete several subproblems while another works on one particularly difficult case. And

a slow or busy processor might complete only one or two subproblems while another processor finishes five or six. Each processor can work at its own pace. As long as the subproblems are fairly small, most of the processors can be kept busy until near the end of the computation. This is known as ***load balancing***: the computational load is balanced among the available processors in order to keep them all as busy as possible. Of course, some processors will still finish before others, but not by longer than the time it takes to complete the longest subtask.

While the subproblems should be small, they should not be **too** small. There is some computational overhead involved in creating the subproblems and assigning them to processors. If the subproblems are very small, this overhead can add significantly to the total amount of work that has to be done. In my example program, the task is to compute a color for each pixel in an image. For dividing that task up into subtasks, one possibility would be to have each subtask compute just one pixel. But the subtasks produced in that way are probably too small. So, instead, each subtask in my program will compute the colors for one row of pixels. Since there are several hundred rows of pixels in the image, the number of subtasks will be fairly large, while each subtask will also be fairly large. The result is fairly good load balancing, with a reasonable amount of overhead.

Note, by the way, that the problem that we are working on is a very easy one for parallel programming. When we divide the problem of calculating an image into subproblems, all the subproblems are completely independent. It is possible to work on any number of them simultaneously, and they can be done in any order. Things get a lot more complicated when some subtasks produce results that are required by other subtasks. In that case, the subtasks are not independent, and the order in which the subtasks are performed is important. Furthermore, there has to be some way for results from one subtask to be shared with other tasks. When the subtasks are executed by different threads, this raises all the issues involved in controlling access of threads to shared resources. So, in general, decomposing a problem for parallel processing is much more difficult than it might appear from our relatively simple example.

12.3.2 Thread Pools and Task Queues

Once we have decided how to decompose a task into subtasks, there is the question of how to assign those subtasks to threads. Typically, in an object-oriented approach, each subtask will be represented by an object. Since a task represents some computation, it's natural for the object that represents it to have an instance method that does the computation. To execute the task, it is only necessary to call its computation method. In my program, the computation method is called `run()` and the task object implements the standard *Runnable* interface that was discussed in Subsection 12.1.1. This interface is a natural way to represent computational tasks. It's possible to create a new thread for each *Runnable*. However, that doesn't really make sense when there are many tasks, since there is a significant amount of overhead involved in creating each new thread. A better alternative is to create just a few threads and let each thread execute a number of tasks.

The optimal number of threads to use is not entirely clear, and it can depend on exactly what problem you are trying to solve. The goal is to keep all of the computer's processors busy. In the image-computing example, it works well to create one thread for each available processor, but that won't be true for all problems. In particular, if a thread can block for a non-trivial amount of time while waiting for some event or for access to some resource, you want to have extra threads around for the processor to run while other threads are blocked. We'll encounter exactly that situation when we turn to using threads with networking in Section 12.4.

When several threads are available for performing tasks, those threads are called a ***thread***

pool. Thread pools are used to avoid creating a new thread to perform each task. Instead, when a task needs to be performed, it can be assigned to any idle thread in the "pool."

Once all the threads in the thread pool are busy, any additional tasks will have to wait until one of the threads becomes idle. This is a natural application for a queue: Associated with the thread pool is a queue of waiting tasks. As tasks become available, they are added to the queue. Every time that a thread finishes a task, it goes to the queue to get another task to work on.

Note that there is only one task queue for the thread pool. All the threads in the pool use the same queue, so the queue is a shared resource. As always with shared resources, race conditions are possible and synchronization is essential. Without synchronization, for example, it is possible that two threads trying to get items from the queue at the same time will end up retrieving the same item. (See if you can spot the race conditions in the `dequeue()` method in Subsection 9.3.2.)

Java has a built-in class to solve this problem: *ConcurrentLinkedQueue*. This class and others that can be useful in parallel programming are defined in the package `java.util.concurrent`. It is a parameterized class so that to create, for example, a queue that can hold objects of type *Runnable*, you could say

```
ConcurrentLinkedQueue<Runnable> queue = new ConcurrentLinkedQueue<Runnable>();
```

This class represents a queue, implemented as a linked list, in which operations on the queue are properly synchronized. The operations on a *ConcurrentLinkedQueue* are not exactly the queue operations that we are used to. The method for adding a new item, `x`, to the end of `queue` is `queue.add(x)`. The method for removing an item from the front of `queue` is `queue.poll()`. The `queue.poll()` method returns `null` if the queue is empty; thus, `poll()` can be used to test whether the queue is empty and to retrieve an item if it is not. It makes sense to do things in this way because testing whether the queue is non-empty before taking an item from the queue involves a race condition: Without synchronization, it is possible for another thread to remove the last item from the queue between the time when you check that the queue is non-empty and the time when you try to take the item from the queue. By the time you try to get the item, there's nothing there!

* * *

To use *ConcurrentLinkedQueue* in our image-computing example, we can use the queue along with a thread pool. To begin the computation of the image, we create all the tasks that make up the image and add them to the queue. Then, we can create and start the worker threads that will execute the tasks. Each thread will run in a loop in which it gets one task from the queue, by calling the queue's `poll()` method, and carries out that task. Since the task is an object of type *Runnable*, it is only necessary for the thread to call the task's `run()` method. When the `poll()` method returns `null`, the queue is empty and the thread can terminate because all the tasks have been assigned to threads.

The sample program *MultiprocessingDemo2.java* implements this idea. It uses a queue `taskQueue` of type *ConcurrentLinkedQueue<Runnable>* to hold the tasks. In addition, in order to allow the user to abort the computation before it finishes, it uses the `volatile` **boolean** variable `running` to signal the thread when the user aborts the computation. The thread should terminate when this variable is set to `false`. The threads are defined by a nested class named *WorkerThread*. It is quite short and simple to write at this point:

```
private class WorkerThread extends Thread {
    public void run() {
```

```
            try {
                while (running) {
                    Runnable task = taskQueue.poll(); // Get a task from the queue.
                    if (task == null)
                        break; // (because the queue is empty)
                    task.run();  // Execute the task;
                }
            }
            finally {
                threadFinished(); // Records fact that this thread has terminated.
            }
        }
    }
```

The program uses a nested class named `MandelbrotTask` to represent the task of computing one row of pixels in the image. This class implements the *Runnable* interface. Its `run()` method does the actual work: Compute the color of each pixel, and apply the colors to the image. Here is what the program does to start the computation (with a few details omitted):

```
taskQueue = new ConcurrentLinkedQueue<Runnable>(); // Create the queue.
int height = ... ; // Number of rows in the image.
for (int row = 0; row < height; row++) {
    MandelbrotTask task;
    task = ... ;  // Create a task to compute one row of the image.
    taskQueue.add(task); // Add the task to the queue.
}

int threadCount = ... ; // Number of threads in the pool
workers = new WorkerThread[threadCount];
running = true;  // Set the signal before starting the threads!
threadsCompleted = 0;  // Records how many of the threads have terminated.
for (int i = 0; i < threadCount; i++) {
    workers[i] = new WorkerThread();
    try {
        workers[i].setPriority( Thread.currentThread().getPriority() - 1 );
    }
    catch (Exception e) {
    }
    workers[i].start();
}
```

Note that it is important that the tasks be added to the queue **before** the threads are started. The threads see an empty queue as a signal to terminate. If the queue is empty when the threads are created, they might see an empty queue and terminate immediately after being started, without performing any tasks!

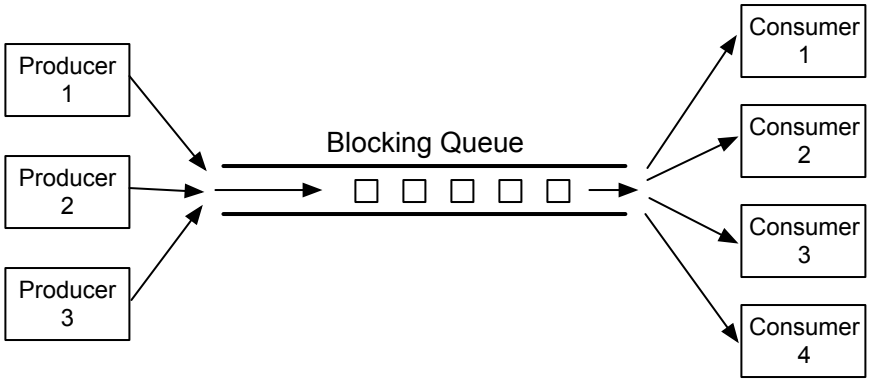
You should run `MultiprocessingDemo2` or try the applet version in the on-line version of this section. It computes the same image as `MultiprocessingDemo1`, but the rows of pixels are not computed in the same order as in that program (assuming that there is more than one thread). If you look carefully, you might see that the rows of pixels are not added to the image in strict order from top to bottom. This is because it is possible for one thread to finish row number `i+1` while another thread is still working on row `i`, or even earlier rows. (The effect might be more apparent if you use more threads than you have processors.)

12.3.3 Producer/Consumer and Blocking Queues

`MultiprocessingDemo2` creates an entirely new thread pool every time it draws an image. This seems wasteful. Shouldn't it be possible to create one set of threads at the beginning of the program and use them whenever an image needs to be computed? After all, the idea of a thread pool is that the threads should sit around and wait for tasks to come along and should execute them when they do. The problem is that, so far, we have no way to make a thread *wait* for a task to come along. To do that, we will use something called a ***blocking queue***.

A blocking queue is an implementation of one of the classic patterns in parallel processing: the ***producer/consumer*** pattern. This pattern arises when there are one or more "producers" who produce things and one or more "consumers" who consume those things. All the producers and consumers should be able to work simultaneously (hence, parallel processing). If there are no things ready to be processed, a consumer will have to wait until one is produced. In many applications, producers also have to wait sometimes: If things can only be consumed at a rate of, say, one per minute, it doesn't make sense for the producers to produce them indefinitely at a rate of two per minute. That would just lead to an unlimited build-up of things waiting to be processed. Therefore, it's often useful to put a limit on the number of things that can be waiting for processing. When that limit is reached, producers should wait before producing more things.

We need a way to get the things from the producers to the consumers. A queue is an obvious answer: Producers can place items into the queue as they are produced. Consumers can remove items from the other end of the queue.



We are talking parallel processing, so we need a synchronized queue, but we need more than that. When the queue is empty, we need a way to have consumers *wait* until an item appears in the queue. If the queue becomes full, we need a way to have producers *wait* until a space opens up in the queue. In our application, the producers and consumers are threads. A thread that is suspended, waiting for something to happen, is said to be blocked, and the type of queue that we need is called a blocking queue. In a blocking queue, the operation of dequeueing an item from the queue can block if the queue is empty. That is, if a thread tries to dequeue an item from an empty queue, the thread will be suspended until an item becomes available; at that time, it will wake up, retrieve the item, and proceed. Similarly, if the queue has a limited capacity, a producer that tries to enqueue an item can block if there is no space in the queue.

Java has two classes that implement blocking queues: *LinkedBlockingQueue* and *ArrayBlockingQueue*. These are parameterized types to allow you to specify the type of item that the queue can hold. Both classes are defined in the package `java.util.concurrent` and both implement

an interface called *BlockingQueue*. If `bqueue` is a blocking queue belonging to one of these classes, then the following operations are defined:

- `bqueue.take()` – Removes an item from the queue and returns it. If the queue is empty when this method is called, the thread that called it will block until an item becomes available. This method throws an *InterruptedException* if the thread is interrupted while it is blocked.
- `bqueue.put(item)` – Adds the `item` to the queue. If the queue has a limited capacity and is full, the thread that called it will block until a space opens up in the queue. This method throws an *InterruptedException* if the thread is interrupted while it is blocked.
- `bqueue.add(item)` – Adds the `item` to the queue, if space is available. If the queue has a limited capacity and is full, an *IllegalStateException* is thrown. This method does not block.
- `bqueue.clear()` – Removes all items from the queue and discards them.

Java's blocking queues define many additional methods (for example, `bqueue.poll(500)` is similar to `bqueue.take()`, except that it will not block for longer than 500 milliseconds), but the four listed here are sufficient for our purposes. Note that I have listed two methods for adding items to the queue: `bqueue.put(item)` blocks if there is not space available in the queue and is meant for use with blocking queues that have a limited capacity; `bqueue.add(item)` does not block and is meant for use with blocking queues that have an unlimited capacity.

An *ArrayBlockingQueue* has a maximum capacity that is specified when it is constructed. For example, to create a blocking queue that can hold up to 25 objects of type *ItemType*, you could say:

```
ArrayBlockingQueue<ItemType> bqueue = new ArrayBlockingQueue<ItemType>(25);
```

With this declaration, `bqueue.put(item)` will block if `bqueue` already contains 25 items, while `bqueue.add(item)` will throw an exception in that case. Recall that this ensures that tasks are not produced indefinitely at a rate faster than they can be consumed. A *LinkedBlockingQueue* is meant for creating blocking queues with unlimited capacity. For example,

```
LinkedBlockingQueue<ItemType> bqueue = new LinkedBlockingQueue<ItemType>();
```

creates a queue with no upper limit on the number of items that it can contain. In this case, `bqueue.put(item)` will never block and `bqueue.add(item)` will never throw an `IllegalStateException`. You would use a *LinkedBlockingQueue* when you want to avoid blocking, and you have some other way of ensuring that the queue will not grow to arbitrary size. For both types of blocking queue, `bqueue.take()` will block if the queue is empty.

* * *

The sample program *MultiprocessingDemo3.java* uses a *LinkedBlockingQueue* in place of the *ConcurrentLinkedQueue* in the previous version, *MultiprocessingDemo2.java*. In this example, the queue holds tasks, that is, items of type *Runnable*, and the queue is declared as an instance variable named `taskQueue`:

```
LinkedBlockingQueue<Runnable> taskQueue;
```

When the user clicks the "Start" button and it's time to compute an image, all of the tasks that make up the computation are put into this queue. This is done by calling `taskQueue.add(task)` for each task. It's important that this can be done without blocking, since the tasks are created in the event-handling thread, and we don't want to block that. The queue cannot grow

indefinitely because the program only works on one image at a time, and there are only a few hundred tasks per image.

Just as in the previous version of the program, worker threads belonging to a thread pool will remove tasks from the queue and carry them out. However, in this case, the threads are created once at the beginning of the program—actually, the first time the "Start" button is pressed—and the same threads are reused for any number of images. When there are no tasks to execute, the task queue is empty and the worker threads will block until tasks become available. Each worker thread runs in an infinite loop, processing tasks forever, but it will spend a lot of its time blocked, waiting for a task to be added to the queue. Here is the inner class that defines the worker threads:

```
/**
 * This class defines the worker threads that make up the thread pool.
 * A WorkerThread runs in a loop in which it retrieves a task from the
 * taskQueue and calls the run() method in that task.  Note that if
 * the queue is empty, the thread blocks until a task becomes available
 * in the queue.  The constructor starts the thread, so there is no
 * need for the main program to do so.  The thread will run at a priority
 * that is one less than the priority of the thread that calls the
 * constructor.
 *
 * A WorkerThread is designed to run in an infinite loop.  It will
 * end only when the Java virtual machine exits. (This assumes that
 * the tasks that are executed don't throw exceptions, which is true
 * in this program.)  The constructor sets the thread to run as
 * a daemon thread; the Java virtual machine will exit when the
 * only threads are daemon threads.  (In this program, this is not
 * necessary since the virtual machine is set to exit when the
 * window is closed.  In a multi-window program, however, we can't
 * simply end the program when a window is closed.)
 */
private class WorkerThread extends Thread {
    WorkerThread() {
        try {
            setPriority( Thread.currentThread().getPriority() - 1);
        }
        catch (Exception e) {
        }
        try {
            setDaemon(true);
        }
        catch (Exception e) {
        }
        start();
    }
    public void run() {
        while (true) {
            try {
                Runnable task = taskQueue.take(); // wait for task if necessary
                task.run();
            }
            catch (InterruptedException e) {
            }
```

```
            }
        }
    }
```

We should look more closely at how the thread pool works. The worker threads are created and started before there is any task to perform. Each thread immediately calls `taskQueue.take()`. Since the task queue is empty, all the worker threads will block as soon as they are started. To start the computation of an image, the event-handling thread will create tasks and add them to the queue. As soon as this happens, worker threads will wake up and start processing tasks, and they will continue doing so until the queue is emptied. (Note that on a multi-processor computer, some worker threads can start processing even while the event thread is still adding tasks to the queue.) When the queue is empty, the worker threads will go back to sleep until processing starts on the next image.

* * *

An interesting point in this program is that we want to be able to abort the computation before it finishes, but we don't want the worker threads to terminate when that happens. When the user clicks the "Abort" button, the program calls `taskQueue.clear()`, which prevents any more tasks from being assigned to worker threads. However, some tasks are most likely already being executed when the task queue is cleared. Those tasks will complete **after** the computation in which they are subtasks has supposedly been aborted. When those subtasks complete, we don't want their output to be applied to the image. It's not a big deal in this program, but in more general applications, we don't want output meant for a previous computation job to be applied to later jobs.

My solution is to assign a job number to each computation job. The job number of the current job is stored in an instance variable named `jobNum`, and each task object has an instance variable that tells which task that job is part of. When a job ends—either because the job finishes on its own or because the user aborts it—the value of `jobNum` is incremented. When a task completes, the job number stored in the task object is compared to `jobNum`. If they are equal, then the task is part of the current job, and its output is applied to the image. If they are not equal, then the task was part of a previous job, and its output is discarded.

It's important that access to `jobNum` be properly synchronized. Otherwise, one thread might check the job number just as another thread is incrementing it, and output meant for an old job might sneak through after that job has been aborted. In the program, all the methods that access or change `jobNum` are synchronized. You can read the *source code* to see how it works.

* * *

One more point about `MultiprocessingDemo3`.... I have not provided any way to terminate the worker threads in this program. They will continue to run until the Java Virtual Machine exits. To allow thread termination before that, we could use a `volatile` signaling variable, `running`, and set its value to `false` when we want the worker threads to terminate. The `run()` methods for the threads would be replaced by

```
public void run() {
    while ( running ) {
       try {
          Runnable task = taskQueue.take();
          task.run();
       }
       catch (InterruptedException e) {
       }
```

```
        }
    }
```

However, if a thread is blocked in `taskQueue.take()`, it will not see the new value of `running` until it becomes unblocked. To ensure that that happens, it is necessary to call `worker.interrupt()` for each worker thread `worker`, just after setting `runner` to `false`.

If a worker thread is executing a task when `runner` is set to `false`, the thread will not terminate until that task has completed. If the tasks are reasonably short, this is not a problem. If tasks can take longer to execute than you are willing to wait for the threads to terminate, then each task must also check the value of `running` periodically and exit when that value becomes `false`.

12.3.4 Wait and Notify

To implement a blocking queue, we must be able to make a thread block just until some event occurs. The thread is *waiting* for the event to occur. Somehow, it must be *notified* when that happens. There are two threads involved since the event that will wake one thread is caused by an action taken by another thread, such as adding an item to the queue.

Note that this is not just an issue for blocking queues. Whenever one thread produces some sort of result that is needed by another thread, that imposes some restriction on the order in which the threads can do their computations. If the second thread gets to the point where it needs the result from the first thread, it might have to stop and wait for the result to be produced. Since the second thread can't continue, it might as well go to sleep. But then there has to be some way to notify the second thread when the result is ready, so that it can wake up and continue its computation.

Java, of course, has a way to do this kind of "waiting" and "notifying": It has `wait()` and `notify()` methods that are defined as instance methods in class *Object* and so can be used with any object. These methods are used internally in blocking queues. They are fairly low-level, tricky, and error-prone, and you should use higher-level control strategies such as blocking queues when possible. However, it's nice to know about `wait()` and `notify()` in case you ever need to use them directly.

The reason why `wait()` and `notify()` should be associated with objects is not obvious, so don't worry about it at this point. It does, at least, make it possible to direct different notifications to different recipients, depending on which object's `notify()` method is called.

The general idea is that when a thread calls a `wait()` method in some object, that thread goes to sleep until the `notify()` method in the **same** object is called. It will have to be called, obviously, by another thread, since the thread that called `wait()` is sleeping. A typical pattern is that Thread A calls `wait()` when it needs a result from Thread B, but that result is not yet available. When Thread B has the result ready, it calls `notify()`, which will wake Thread A up, if it is waiting, so that it can use the result. It is not an error to call `notify()` when no one is waiting; it just has no effect. To implement this, Thread A will execute code similar to the following, where `obj` is some object:

```
if ( resultIsAvailable() == false )
   obj.wait();  // wait for notification that the result is available
useTheResult();
```

while Thread B does something like:

```
generateTheResult();
obj.notify();  // send out a notification that the result is available
```

Now, there is a really nasty race condition in this code. The two threads might execute their code in the following order:

```
1.  Thread A checks resultIsAvailable() and finds that the result is not ready,
        so it decides to execute the obj.wait() statement, but before it does,
2.  Thread B finishes generating the result and calls obj.notify()
3.  Thread A calls obj.wait() to wait for notification that the result is ready.
```

In Step 3, Thread A is waiting for a notification that will never come, because `notify()` has already been called in Step 2. This is a kind of deadlock that can leave Thread A waiting forever. Obviously, we need some kind of synchronization. The solution is to enclose both Thread A's code and Thread B's code in `synchronized` statements, and it is very natural to synchronize on the same object, `obj`, that is used for the calls to `wait()` and `notify()`. In fact, since synchronization is almost always needed when `wait()` and `notify()` are used, Java makes it an absolute requirement. In Java, a thread can legally call `obj.wait()` or `obj.notify()` **only** if that thread holds the synchronization lock associated with the object `obj`. If it does not hold that lock, then an exception is thrown. (The exception is of type *IllegalMonitorStateException*, which does not require mandatory handling and which is typically not caught.) One further complication is that the `wait()` method can throw an *InterruptedException* and so should be called in a `try` statement that handles the exception.

To make things more definite, let's consider how we can get a result that is computed by one thread to another thread that needs the result. This is a simplified producer/consumer problem in which only one item is produced and consumed. Assume that there is a shared variable named `sharedResult` that is used to transfer the result from the producer to the consumer. When the result is ready, the producer sets the variable to a non-null value. The consumer can check whether the result is ready by testing whether the value of `sharedResult` is null. We will use a variable named `lock` for synchronization. The code for the producer thread could have the form:

```
makeResult = generateTheResult();  // Not synchronized!
synchronized(lock) {
   sharedResult = makeResult;
   lock.notify();
}
```

while the consumer would execute code such as:

```
synchronized(lock) {
   while ( sharedResult == null ) {
      try {
         lock.wait();
      }
      catch (InterruptedException e) {
      }
   }
   useResult = sharedResult;
}
useTheResult(useResult);  // Not synchronized!
```

The calls to `generateTheResult()` and `useTheResult()` are not synchronized, which allows them to run in parallel with other threads that might also synchronize on `lock`. Since `sharedResult` is a shared variable, all references to `sharedResult` should be synchronized, so

the references to `sharedResult` must be inside the `synchronized` statements. The goal is to do as little as possible (but not less) in synchronized code segments.

If you are uncommonly alert, you might notice something funny: `lock.wait()` does not finish until `lock.notify()` is executed, but since both of these methods are called in `synchronized` statements that synchronize on the same object, shouldn't it be impossible for both methods to be running at the same time? In fact, `lock.wait()` is a special case: When a thread calls `lock.wait()`, it gives up the lock that it holds on the synchronization object, `lock`. This gives another thread a chance to execute the `synchronized(lock)` block that contains the `lock.notify()` statement. After the second thread exits from this block, the lock is returned to the consumer thread so that it can continue.

In the full producer/consumer pattern, multiple results are produced by one or more producer threads and are consumed by one or more consumer threads. Instead of having just one `sharedResult` object, we keep a list of objects that have been produced but not yet consumed. Let's see how this might work in a very simple class that implements the three operations on a *LinkedBlockingQueue<Runnable>* that are used in `MultiprocessingDemo3`:

```
import java.util.LinkedList;

public class MyLinkedBlockingQueue {

    private LinkedList<Runnable> taskList = new LinkedList<Runnable>();

    public void clear() {
        synchronized(taskList) {
            taskList.clear();
        }
    }

    public void add(Runnable task) {
        synchronized(taskList) {
            taskList.addLast(task);
            taskList.notify();
        }
    }

    public Runnable take() throws InterruptedException {
        synchronized(taskList) {
            while (taskList.isEmpty())
                taskList.wait();
            return taskList.removeFirst();
        }
    }

}
```

An object of this class could be used as a direct replacement for the `taskQueue` in `MultiprocessingDemo3`.

In this class, I have chosen to synchronize on the `taskList` object, but any object could be used. In fact, I could simply use `synchronized` methods, which is equivalent to synchronizing on `this`. (Note that you might see a call to `wait()` or `notify()` in a `synchronized` instance method, with no reference to the object that is being used. Remember that `wait()` and `notify()` in that context really mean `this.wait()` and `this.notify()`.)

By the way, it is essential that the call to `taskList.clear()` be synchronized on the same object, even though it doesn't call `wait()` or `notify()`. Otherwise, there is a race condition

that can occur: The list might be cleared just after the `take()` method checks that `taskList` is non-empty and before it removes an item from the list. In that case, the list is empty again by the time `taskList.removeFirst()` is called, resulting in an error.

* * *

It is possible for several threads to be waiting for notification. A call to `obj.notify()` will wake only one of the threads that is waiting on `obj`. If you want to wake all threads that are waiting on `obj`, you can call `obj.notifyAll()`. `obj.notify()` works OK in the above example because only consumer threads can be blocked. We only need to wake one consumer thread when a task is added to the queue because it doesn't matter which consumer gets the task. But consider a blocking queue with limited capacity, where producers and consumers can both block. When an item is added to the queue, we want to make sure that a consumer thread is notified, not just another producer. One solution is to call `notifyAll()` instead of `notify()`, which will notify all threads including any waiting consumer.

I should also mention a possible confusion about the method `obj.notify()`. This method does **not** notify `obj` of anything. It notifies a thread that has called `obj.wait()` (if there is such a thread). Similarly, in `obj.wait()`, it's **not** `obj` that is waiting for something; it's the thread that calls the method.

And a final note on `wait`: There is another version of `wait()` that takes a number of milliseconds as a parameter. A thread that calls `obj.wait(milliseconds)` will wait only up to the specified number of milliseconds for a notification. If a notification doesn't occur during that period, the thread will wake up and continue without the notification. In practice, this feature is most often used to let a waiting thread wake periodically while it is waiting in order to perform some periodic task, such as causing a message "Waiting for computation to finish" to blink.

* * *

Let's look at an example that uses `wait()` and `notify()` to allow one thread to control another. The sample program *TowersOfHanoiWithControls.java* solves the Towers Of Hanoi puzzle (Subsection 9.1.2), with control buttons that allow the user to control the execution of the algorithm. Clicking "Next Step" executes one step, which moves a single disk from one pile to another. Clicking "Run" lets the algorithm run automatically on its own; "Run" changes to "Pause", and clicking "Pause" stops the automatic execution. There is also a "Start Over" button that aborts the current solution and puts the puzzle back into its initial configuration. Here is a picture of the program in the middle of a solution:

In this program, there are two threads: a thread that runs a recursive algorithm to solve the puzzle, and the event-handling thread that reacts to user actions. When the user clicks one of the buttons, a method is called in the event-handling thread. But it's actually the thread that is running the recursion that has to respond by, for example, doing one step of the solution or starting over. The event-handling thread has to send some sort of signal to the solution thread.

This is done by setting the value of a variable that is shared by both threads. The variable is named `status`, and its possible values are the constants `GO`, `PAUSE`, `STEP`, and `RESTART`.

When the event-handling thread changes the value of this variable, the solution thread should see the new value and respond. When `status` equals `PAUSE`, the solution thread is paused, waiting for the user to click "Run" or "Next Step". This is the initial state, when the program starts. If the user clicks "Next Step", the event-handling thread sets the value of `status` to "STEP"; the solution thread should respond by executing one step of the solution and then resetting `status` to `PAUSE`. If the user clicks "Run", `status` is set to `GO`, which should cause the solution thread to run automatically. When the user clicks "Pause" while the solution is running, `status` is reset to `PAUSE`, and the solution thread should return to its paused state. If the user clicks "Start Over", the event-handling thread sets `status` to `RESTART`, and the solution thread should respond by ending the current recursive solution and restoring the puzzle to its initial state.

The main point for us is that when the solution thread is paused, it is *sleeping*. It won't see a new value for `status` unless it wakes up! To make that possible, the program uses `wait()` in the solution thread to put that thread to sleep, and it uses `notify()` in the event-handling thread to wake up the solution thread whenever it changes the value of `status`. Here is the `actionPerformed()` method that responds to clicks on the buttons. When the user clicks a button, this method changes the value of `status` and calls `notify()` to wake up the solution thread:

```
synchronized public void actionPerformed(ActionEvent evt) {
    Object source = evt.getSource();
    if (source == runPauseButton) {  // Toggle between running and paused.
        if (status == GO) {  // Animation is running.  Pause it.
            status = PAUSE;
            nextStepButton.setEnabled(true);  // Enable while paused.
            runPauseButton.setText("Run");
        }
        else {  // Animation is paused.  Start it running.
            status = GO;
            nextStepButton.setEnabled(false);  // Disable while running.
            runPauseButton.setText("Pause");
        }
    }
    else if (source == nextStepButton) {  // Makes animation run one step.
        status = STEP;
    }
    else if (source == startOverButton) { // Restore to initial state.
        status = RESTART;
    }
    notify();  // Wake up the thread so it can see the new status value!
}
```

This method is synchronized to allow the call to `notify()`. Remember that the `notify()` method in an object can only be called by a thread that holds that object's synchronization lock. In this case, the synchronization object is `this`. Synchronization is also necessary because of race conditions that arise because the value of `status` can also be changed by the solution thread.

The solution thread calls a method named `checkStatus()` to check the value of `status`. This method calls `wait()` if the status is `PAUSE`, which puts the solution thread to sleep until

the event-handling thread calls `notify()`. Note that if the status is `RESTART`, `checkStatus()` throws an *IllegalStateException*:

```
synchronized private void checkStatus() {
    while (status == PAUSE) {
        try {
            wait();
        }
        catch (InterruptedException e) {
        }
    }
    // At this point, status is RUN, STEP, or RESTART.
    if (status == RESTART)
        throw new IllegalStateException("Restart");
    // At this point, status is RUN or STEP.
}
```

The `run()` method for the solution thread runs in an infinite loop in which it sets up the initial state of the puzzle and then calls a `solve()` method to solve the puzzle. To implement the wait/notify control strategy, `run()` calls `checkStatus()` before starting the solution, and `solve()` calls `checkStatus()` after each move. If `checkStatus()` throws an *IllegalStateException*, the call to `solve()` is terminated early, and the `run()` method returns to the beginning of the while loop, where the initial state of the puzzle, and of the user interface, is restored:

```
public void run() {
    while (true) {
        runPauseButton.setText("Run");   // Set user interface to initial state.
        nextStepButton.setEnabled(true);
        startOverButton.setEnabled(false);
        setUpProblem();  // Set up the initial state of the puzzle
        status = PAUSE;  // Initially, the solution thread is paused.
        checkStatus(); // Returns only when user has clicked "Run" or "Next Step"
        startOverButton.setEnabled(true);
        try {
            solve(10,0,1,2);  // Move 10 disks from pile 0 to pile 1.
        }
        catch (IllegalStateException e) {
            // Exception was thrown because use clicked "Start Over".
        }
    }
}
```

You can check the *source code* to see how this all fits into the complete program. If you want to learn how to use `wait()` and `notify()` directly, understanding this example is a good place to start!

12.4 Threads and Networking

In the previous chapter, we looked at several examples of network programming. Those examples showed how to create network connections and communicate through them, but they didn't deal with one of the fundamental characteristics of network programming, the fact that network communication is asynchronous. From the point of view of a program on one end of a network connection, messages can arrive from the other side of the connection at any time; the

arrival of a message is an *event* that is not under the control of the program that is receiving the message. Perhaps an event-oriented networking API would be a good approach to dealing with the asynchronous nature of network communication, but that is not the approach that is taken in Java (or, typically, in other languages). Instead, network programming in Java typically uses **threads**.

12.4.1 The Blocking I/O Problem

A covered in Section 11.4, network programming uses sockets. A socket, in the sense that we are using the term here, represents one end of a network connection. Every socket has an associated input stream and output stream. Data written to the output stream on one end of the connection is transmitted over the network and appears in the input stream at the other end.

A program that wants to read data from a socket's input stream calls one of that input stream's input methods. It is possible that the data has already arrived before the input method is called; in that case, the input method retrieves the data and returns immediately. More likely, however, the input method will have to wait for data to arrive from the other side of the connection. Until the data arrives, the input method and the thread that called it will be blocked.

It is also possible for an output method in a socket's output stream to block. This can happen if the program tries to output data to the socket faster than the data can be transmitted over the network. (It's a little complicated: a socket uses a "buffer" to hold data that is supposed to be transmitted over the network. A buffer is just a block of memory that is used like a queue. The output method drops its data into the buffer; lower-level software removes data from the buffer and transmits it over the network. The output method will block if the buffer fills up. Note that when the output method returns, it doesn't mean that the data has gone out over the network—it just means that the data has gone into the buffer and is scheduled for later transmission.)

We say that network communication uses ***blocking I/O***, because input and output operations on the network can block for indefinite periods of time. Programs that use the network must be prepared to deal with this blocking. In some cases, it's acceptable for a program to simply shut down all other processing and wait for input. (This is what happens when a command line program reads input typed by the user. User input is another type of blocking I/O.) However, threads make it possible for some parts of a program to continue doing useful work while other parts are blocked. A network client program that sends requests to a server might get by with a single thread, if it has nothing else to do while waiting for the server's responses. A network server program, on the other hand, can typically be connected to several clients at the same time. While waiting for data to arrive from a client, the server certainly has other things that it can do, namely communicate with other clients. When a server uses different threads to handle the communication with different clients, the fact that I/O with one client is blocked won't stop the server from communicating with other clients.

It's important to understand that using threads to deal with blocking I/O differs in a fundamental way from using threads to speed up computation. When using threads for speed-up in Subsection 12.3.2, it made sense to use one thread for each available processor. If only one processor is available, using more than one thread will yield no speed-up at all; in fact, it would slow things down because of the extra overhead involved in creating and managing the threads.

In the case of blocking I/O, on the other hand, it can make sense to have many more threads

than there are processors, since at any given time many of the threads can be blocked. Only the active, unblocked threads are competing for processing time. In the ideal case, to keep all the processors busy, you would want to have one **active** thread per processor (actually somewhat less than that, on average, to allow for variations over time in the number of active threads). On a network server program, for example, threads generally spend **most** of their time blocked waiting for I/O operations to complete. If threads are blocked, say, about 90% of the time, you'd like to have about ten times as many threads as there are processors. So even on a computer that has just a single processor, server programs can make good use of large numbers of threads.

12.4.2 An Asynchronous Network Chat Program

As a first example of using threads for network communication, we consider a GUI chat program.

The command-line chat programs, *CLChatClient.java* and *CLChatServer.java*, from the Subsection 11.4.5 use a straight-through, step-by-step protocol for communication. After a user on one side of a connection enters a message, the user must wait for a reply from the other side of the connection. An asynchronous chat program would be much nicer. In such a program, a user could just keep typing lines and sending messages without waiting for any response. Messages that arrive—asynchronously—from the other side would be displayed as soon as they arrive. It's not easy to do this in a command-line interface, but it's a natural application for a graphical user interface. The basic idea for a GUI chat program is to create a thread whose job is to read messages that arrive from the other side of the connection. As soon as the message arrives, it is displayed to the user; then, the message-reading thread blocks until the next incoming message arrives. While it is blocked, however, other threads can continue to run. In particular, the event-handling thread that responds to user actions keeps running; that thread can send outgoing messages as soon as the user generates them.

The sample program *GUIChat.java* is an example of this. `GUIChat` is a two-way network chat program that allows two users to send messages to each other over the network. In this chat program, each user can send messages at any time, and incoming messages are displayed as soon as they are received.

The `GUIChat` program can act as either the client end or the server end of a connection. (Refer back to Subsection 11.4.3 for information about how clients and servers work.) The program has a "Listen" button that the user can click to create a server socket that will listen for an incoming connection request; this makes the program act as a server. It also has a "Connect" button that the user can click to send a connection request; this makes the program act as a client. As usual, the server listens on a specified port number. The client needs to know the computer on which the server is running and the port on which the server is listening. There are input boxes in the `GUIChat` window where the user can enter this information.

Once a connection has been established between two `GUIChat` windows, each user can send messages to the other. The window has an input box where the user types a message. Pressing return while typing in this box sends the message. This means that the sending of the message is handled by the usual event-handling thread, in response to an event generated by a user action. Messages are received by a separate thread that just sits around waiting for incoming messages. This thread blocks while waiting for a message to arrive; when a message does arrive, it displays that message to the user. The window contains a large transcript area that displays both incoming and outgoing messages, along with other information about the network connection.

I urge you to compile the source code, *GUIChat.java*, and try the program. To make it easy

to try it on a single computer, you can make a connection between one window and another window on the same computer, using "localhost" or "127.0.0.1" as the name of the computer. (Once you have one `GUIChat` window open, you can open a second one by clicking the "New" button.) I also urge you to read the source code. I will discuss only parts of it here.

The program uses a nested class, *ConnectionHandler*, to handle most network-related tasks. *ConnectionHandler* is a subclass of *Thread*. The *ConnectionHandler* thread is responsible for opening the network connection and then for reading incoming messages once the connection has been opened. By putting the connection-opening code in a separate thread, we make sure that the GUI is not blocked while the connection is being opened. Like reading incoming messages, opening a connection is a blocking operation that can take some time to complete. A *ConnectionHandler* is created when the user clicks the "Listen" or "Connect" button. The "Listen" button should make the thread act as a server, while "Connect" should make it act as a client. To distinguish these two cases, the *ConnectionHandler* class has two constructors:

```
/**
 * Listen for a connection on a specified port.  The constructor
 * does not perform any network operations; it just sets some
 * instance variables and starts the thread.  Note that the
 * thread will only listen for one connection, and then will
 * close its server socket.
 */
ConnectionHandler(int port) {
   state = ConnectionState.LISTENING;
   this.port = port;
   postMessage("\nLISTENING ON PORT " + port + "\n");
   start();
}

/**
 * Open a connection to a specified computer and port.  The constructor
 * does not perform any network operations; it just sets some
 * instance variables and starts the thread.
 */
ConnectionHandler(String remoteHost, int port) {
   state = ConnectionState.CONNECTING;
   this.remoteHost = remoteHost;
   this.port = port;
   postMessage("\nCONNECTING TO " + remoteHost + " ON PORT " + port + "\n");
   start();
}
```

Here, `state` is an instance variable whose type is defined by an enumerated type

```
enum ConnectionState { LISTENING, CONNECTING, CONNECTED, CLOSED };
```

The values of this `enum` represent different possible states of the network connection. It is often useful to treat a network connection as a state machine (see Subsection 6.5.4), since the response to various events can depend on the state of the connection when the event occurs. Setting the `state` variable to `LISTENING` or `CONNECTING` tells the thread whether it should act as a server or as a client. Note that the `postMessage()` method posts a message to the transcript area of the window, where it will be visible to the user.

Once the thread has been started, it executes the following `run()` method:

```
/**
 * The run() method that is executed by the thread.  It opens a
 * connection as a client or as a server (depending on which
 * constructor was used).
 */
public void run() {
   try {
      if (state == ConnectionState.LISTENING) {
            // Open a connection as a server.
         listener = new ServerSocket(port);
         socket = listener.accept();
         listener.close();
      }
      else if (state == ConnectionState.CONNECTING) {
            // Open a connection as a client.
         socket = new Socket(remoteHost,port);
      }
      connectionOpened();  // Sets up to use the connection (including
                           // creating a BufferedReader, in, for reading
                           // incoming messages).
      while (state == ConnectionState.CONNECTED) {
            // Read one line of text from the other side of
            // the connection, and report it to the user.
         String input = in.readLine();
         if (input == null)
            connectionClosedFromOtherSide();
         else
            received(input);  // Report message to user.
      }
   }
   catch (Exception e) {
         // An error occurred.  Report it to the user, but not
         // if the connection has been closed (since the error
         // might be the expected error that is generated when
         // a socket is closed).
      if (state != ConnectionState.CLOSED)
         postMessage("\n\n ERROR:  " + e);
   }
   finally {  // Clean up before terminating the thread.
      cleanUp();
   }
}
```

This method calls several other methods to do some of its work, but you can see the general outline of how it works. After opening the connection as either a server or client, the `run()` method enters a `while` loop in which it receives and processes messages from the other side of the connection until the connection is closed. It is important to understand how the connection can be closed. The `GUIChat` window has a "Disconnect" button that the user can click to close the connection. The program responds to this event by closing the socket that represents the connection. It is likely that when this happens, the connection-handling thread is blocked in the `in.readLine()` method, waiting for an incoming message. When the socket is closed by another thread, this method will fail and will throw an exception; this exception causes the thread to terminate. (If the connection-handling thread happens to be between calls to `in.readLine()`

when the socket is closed, the `while` loop will terminate because the connection state changes from `CONNECTED` to `CLOSED`.) Note that closing the window will also close the connection in the same way.

It is also possible for the user on the other side of the connection to close the connection. When that happens, the stream of incoming messages ends, and the `in.readLine()` on this side of the connection returns the value `null`, which indicates end-of-stream and acts as a signal that the connection has been closed by the remote user.

For a final look into the `GUIChat` code, consider the methods that send and receive messages. These methods are called from different threads. The `send()` method is called by the event-handling thread in response to a user action. Its purpose is to transmit a message to the remote user. (It is conceivable, though not likely, that the data output operation could block, if the socket's output buffer fills up. A more sophisticated program might take this possibility into account.) This method uses a *PrintWriter*, `out`, that writes to the socket's output stream. Synchronization of this method prevents the connection state from changing in the middle of the send operation:

```
/**
 * Send a message to the other side of the connection, and post the
 * message to the transcript.  This should only be called when the
 * connection state is ConnectionState.CONNECTED; if it is called at
 * other times, it is ignored.
 */
synchronized void send(String message) {
   if (state == ConnectionState.CONNECTED) {
      postMessage("SEND:  " + message);
      out.println(message);
      out.flush();
      if (out.checkError()) {
         postMessage("\nERROR OCCURRED WHILE TRYING TO SEND DATA.");
         close();  // Closes the connection.
      }
   }
}
```

The `received()` method is called by the connection-handling thread **after** a message has been read from the remote user. Its only job is to display the message to the user, but again it is synchronized to avoid the race condition that could occur if the connection state were changed by another thread while this method is being executed:

```
/**
 * This is called by the run() method when a message is received from
 * the other side of the connection.  The message is posted to the
 * transcript, but only if the connection state is CONNECTED.  (This
 * is because a message might be received after the user has clicked
 * the "Disconnect" button; that message should not be seen by the
 * user.)
 */
synchronized private void received(String message) {
   if (state == ConnectionState.CONNECTED)
      postMessage("RECEIVE:  " + message);
}
```

12.4.3 A Threaded Network Server

Threads are often used in network server programs. They allow the server to deal with several clients at the same time. When a client can stay connected for an extended period of time, other clients shouldn't have to wait for service. Even if the interaction with each client is expected to be very brief, you can't always assume that that will be the case. You have to allow for the possibility of a misbehaving client—one that stays connected without sending data that the server expects. This can hang up a thread indefinitely, but in a threaded server there will be other threads that can carry on with other clients.

The *DateServer.java* sample program, from Subsection 11.4.4, is an extremely simple network server program. It does not use threads, so the server must finish with one client before it can accept a connection from another client. Let's see how we can turn `DateServer` into a threaded server. (This server is so simple that doing so doesn't make a great deal of sense. However, the same techniques will work for more complicated servers. See, for example, Exercise 12.4.)

As a first attempt, consider *DateServerWithThreads.java*. This sample program creates a new thread every time a connection request is received. The main program simply creates the thread and hands the connection to the thread. This takes very little time, and in particular will not block. The `run()` method of the thread handles the connection in exactly the same way that it would be handled by the original program. This is not at all difficult to program. Here's the new version of the program, with significant changes shown in italic:

```
import java.net.*;
import java.io.*;
import java.util.Date;

/**
 * This program is a server that takes connection requests on
 * the port specified by the constant LISTENING_PORT.  When a
 * connection is opened, the program sends the current time to
 * the connected socket.  The program will continue to receive
 * and process connections until it is killed (by a CONTROL-C,
 * for example).
 *
 * This version of the program creates a new thread for
 * every connection request.
 */
public class DateServerWithThreads {

    public static final int LISTENING_PORT = 32007;

    public static void main(String[] args) {

        ServerSocket listener;  // Listens for incoming connections.
        Socket connection;      // For communication with the connecting program.

        /* Accept and process connections forever, or until some error occurs. */

        try {
            listener = new ServerSocket(LISTENING_PORT);
            System.out.println("Listening on port " + LISTENING_PORT);
            while (true) {
                // Accept next connection request and create thread to handle it.
                connection = listener.accept();
```

```
                ConnectionHandler handler = new ConnectionHandler(connection);
                handler.start();
            }
        }
        catch (Exception e) {
            System.out.println("Sorry, the server has shut down.");
            System.out.println("Error:  " + e);
            return;
        }

    }  // end main()

    /**
     *  Defines a thread that handles the connection with one
     *  client.
     */
    private static class ConnectionHandler extends Thread {
        Socket client;
        ConnectionHandler(Socket socket) {
            client = socket;
        }
        public void run() {
            String clientAddress = client.getInetAddress().toString();
            try {
                System.out.println("Connection from " + clientAddress );
                Date now = new Date();  // The current date and time.
                PrintWriter outgoing;   // Stream for sending data.
                outgoing = new PrintWriter( client.getOutputStream() );
                outgoing.println( now.toString() );
                outgoing.flush();  // Make sure the data is actually sent!
                client.close();
            }
            catch (Exception e){
                System.out.println("Error on connection with: "
                        + clientAddress + ": " + e);
            }
        }
    }

} //end class DateServerWithThreads
```

One interesting change is at the end of the `run()` method, where I've added the `clientAddress` to the output of the error message. I did this to identify which connection the error message refers to. Since threads run in parallel, it's possible for outputs from different threads to be intermingled in various orders. Messages from the same thread don't necessarily come together in the output; they might be separated by messages from other threads. This is just one of the complications that you have to keep in mind when working with threads!

12.4.4 Using a Thread Pool

It's not very efficient to create a new thread for every connection, especially when the connections are typically very short-lived. Fortunately, we have an alternative: thread pools (Subsection 12.3.2).

DateServerWithThreadPool.java is an improved version of our server that uses a thread pool. Each thread in the pool runs in an infinite loop. Each time through the loop, it handles one connection. We need a way for the main program to send connections to the threads. It's natural to use a blocking queue named `connectionQueue` for that purpose. A connection-handling thread takes connections from this queue. Since it is blocking queue, the thread blocks when the queue is empty and wakes up when a connection becomes available in the queue. No other synchronization or communication technique is needed; it's all built into the blocking queue. Here is the `run()` method for the connection-handling threads:

```
public void run() {
    while (true) {
        Socket client;
        try {
            client = connectionQueue.take();  // Blocks until item is available.
        }
        catch (InterruptedException e) {
            continue; // (If interrupted, just go back to start of while loop.)
        }
        String clientAddress = client.getInetAddress().toString();
        try {
            System.out.println("Connection from " + clientAddress );
            System.out.println("Handled by thread " + this);
            Date now = new Date();  // The current date and time.
            PrintWriter outgoing;   // Stream for sending data.
            outgoing = new PrintWriter( client.getOutputStream() );
            outgoing.println( now.toString() );
            outgoing.flush();  // Make sure the data is actually sent!
            client.close();
        }
        catch (Exception e){
            System.out.println("Error on connection with: "
                    + clientAddress + ": " + e);
        }
    }
}
```

The main program, in the meantime, runs in an infinite loop in which connections are accepted and added to the queue:

```
while (true) {
        // Accept next connection request and put it in the queue.
    connection = listener.accept();
    try {
        connectionQueue.put(connection); // Blocks if queue is full.
    }
    catch (InterruptedException e) {
    }
}
```

The queue in this program is of type *ArrayBlockingQueue<Socket>*. As such, it has a limited capacity, and the `put()` operation on the queue will block if the queue is full. But wait—didn't we want to avoid blocking the main program? When the main program is blocked, the server is no longer accepting connections, and clients who are trying to connect are kept waiting. Would it be better to use a *LinkedBlockingQueue*, with an unlimited capacity?

In fact, connections in the queue are waiting anyway; they are not being serviced. If the queue grows unreasonably long, connections in the queue will have to wait for an unreasonable amount of time. If the queue keeps growing indefinitely, that just means that the server is receiving connection requests faster than it can process them. That could happen for several reasons: Your server might simply not be powerful enough to handle the volume of traffic that you are getting; you need to buy a new server. Or perhaps the thread pool doesn't have enough threads to fully utilize your server; you should increase the size of the thread pool to match the server's capabilities. Or maybe your server is under a "Denial Of Service" attack, in which some bad guy is deliberately sending your server more requests than it can handle in an attempt to keep other, legitimate clients from getting service.

In any case, *ArrayBlockingQueue* with limited capacity is the correct choice. The queue should be short enough so that connections in the queue will not have to wait too long for service. In a real server, the size of the queue and the number of threads in the thread pool should be adjusted to "tune" the server to account for the particular hardware and network on which the server is running and for the nature of the client requests that it typically processes. Optimal tuning is, in general, a difficult problem.

There is, by the way, another way that things can go wrong: Suppose that the server needs to read some data from the client, but the client doesn't send the expected data. The thread that is trying to read the data can then block indefinitely, waiting for the input. If a thread pool is being used, this could happen to every thread in the pool. In that case, no further processing can ever take place! The solution to this problem is to have connections "time out" if they are inactive for an excessive period of time. Typically, each connection thread will keep track of the time when it last received data from the client. The server runs another thread (sometimes called a "reaper thread", after the Grim Reaper) that wakes up periodically and checks each connection thread to see how long it has been inactive. A connection thread that has been waiting too long for input is terminated, and a new thread is started in its place. The question of how long the timeout period should be is another difficult tuning issue.

12.4.5 Distributed Computing

We have seen how threads can be used to do parallel processing, where a number of processors work together to complete some task. So far, we have assumed that all the processors were inside one multi-processor computer. But parallel processing can also be done using processors that are in different computers, as long as those computers are connected to a network over which they can communicate. This type of parallel processing—in which a number of computers work together on a task and communicate over a network—is called ***distributed computing***.

In some sense, the whole Internet is an immense distributed computation, but here I am interested in how computers on a network can cooperate to solve some computational problem. There are several approaches to distributed computing that are supported in Java. ***RMI*** and ***CORBA*** are standards that enable a program running on one computer to call methods in objects that exist on other computers. This makes it possible to design an object-oriented program in which different parts of the program are executed on different computers. RMI (Remote Method Invocation) only supports communication between Java objects. CORBA (Common Object Request Broker Architecture) is a more general standard that allows objects written in various programming languages, including Java, to communicate with each other. As is commonly the case in networking, there is the problem of locating services (where in this case, a "service" means an object that is available to be called over the network). That is, how can one computer know which computer a service is located on and what port it is listening

on? RMI and CORBA solve this problem using a "request broker"—a server program running at a known location keeps a list of services that are available on other computers. Computers that offer services register those services with the request broker; computers that need services contact the broker to find out where they are located.

RMI and CORBA are complex systems that are not very easy to use. I mention them here because they are part of Java's standard network API, but I will not discuss them further. Instead, we will look at a relatively simple demonstration of distributed computing that uses only basic networking.

The problem that we will consider is the same one that we used in *MultiprocessingDemo1.java*, and its variations, in Section 12.2 and Section 12.3, namely the computation of a complex image. This is an application that uses the simplest type of parallel programming, in which the problem can be broken down into tasks that can be performed independently, with no communication between the tasks. To apply distributed computing to this type of problem, we can use one "master" program that divides the problem into tasks and sends those tasks over the network to "worker" programs that do the actual work. The worker programs send their results back to the master program, which combines the results from all the tasks into a solution of the overall problem. In this context, the worker programs are often called "slaves," and the program uses the so-called ***master/slave*** approach to distributed computing.

The demonstration program is defined by three source code files: *CLMandelbrotMaster.java* defines the master program; *CLMandelbrotWorker.java* defines the worker programs; and *CLMandelbrotTask.java* defines the class, *CLMandelbrotTask*, that represents an individual task that is performed by the workers. To run the demonstration, you must start the `CLMandelbrotWorker` program on several computers (probably by running it on the command line). This program uses *CLMandelbrotTask*, so both class files, `CLMandelbrotWorker.class` and `CLMandelbrotTask.class`, must be present on the worker computers. You can then run `CLMandelbrotMaster` on the master computer. Note that this program also requires the class *CLMandelbrotTask*. You must specify the host name or IP address of each of the worker computers as command line arguments for `CLMandelbrotMaster`. A worker program listens for connection requests from the master program, and the master program must be told where to send those requests. For example, if the worker program is running on three computers with IP addresses 172.30.217.101, 172.30.217.102, and 172.30.217.103, then you can run `CLMandelbrotMaster` with the command

```
java  CLMandelbrotMaster  172.30.217.101  172.30.217.102  172.30.217.103
```

The master will make a network connection to the worker at each IP address; these connections will be used for communication between the master program and the workers.

It is possible to run several copies of `CLMandelbrotWorker` on the same computer, but they must listen for network connections on different ports. It is also possible to run `CLMandelbrotWorker` on the same computer as `CLMandelbrotMaster`. You might even see some speed-up when you do this, if your computer has several processors. See the comments in the program source code files for more information, but here are some commands that you can use to run the master program and two copies of the worker program on the same computer. Give these commands in separate command windows:

```
java  CLMandelbrotWorker                                (Listens on default port)

java  CLMandelbrotWorker  2501                          (Listens on port 2501)

java  CLMandelbrotMaster  localhost  localhost:2501
```

Every time `CLMandelbrotMaster` is run, it solves exactly the same problem. (For this demonstration, the nature of the problem is not important, but the problem is to compute the data needed for a picture of a small piece of the famous "Mandelbrot Set." If you are interested in seeing the picture that is produced, uncomment the call to the `saveImage()` method at the end of the `main()` routine in *CLMandelbrotMaster.java*.)

You can run `CLMandelbrotMaster` with different numbers of worker programs to see how the time required to solve the problem depends on the number of workers. (Note that the worker programs continue to run after the master program exists, so you can run the master program several times without having to restart the workers.) In addition, if you run `CLMandelbrotMaster` with no command line arguments, it will solve the entire problem on its own, so you can see how long it takes to do so without using distributed computing. In a trial that I ran on some rather old, slow computers, it took 40 seconds for `CLMandelbrotMaster` to solve the problem on its own. Using just one worker, it took 43 seconds. The extra time represents extra work involved in using the network; it takes time to set up a network connection and to send messages over the network. Using two workers (on different computers), the problem was solved in 22 seconds. In this case, each worker did about half of the work, and their computations were performed in parallel, so that the job was done in about half the time. With larger numbers of workers, the time continued to decrease, but only up to a point. The master program itself has a certain amount of work to do, no matter how many workers there are, and the total time to solve the problem can never be less than the time it takes for the master program to do its part. In this case, the minimum time seemed to be about five seconds.

* * *

Let's take a look at how this distributed application is programmed. The master program divides the overall problem into a set of tasks. Each task is represented by an object of type *CLMandelbrotTask*. These tasks have to be communicated to the worker programs, and the worker programs must send back their results. Some protocol is needed for this communication. I decided to use character streams. The master encodes a task as a line of text, which is sent to a worker. The worker decodes the text (into an object of type *CLMandelbrotTask*) to find out what task it is supposed to perform. It performs the assigned task. It encodes the results as another line of text, which it sends back to the master program. Finally, the master decodes the results and combines them with the results from other tasks. After all the tasks have been completed and their results have been combined, the problem has been solved.

A worker receives not just one task, but a sequence of tasks. Each time it finishes a task and sends back the result, it is assigned a new task. After all tasks are complete, the worker receives a "close" command that tells it to close the connection. In *CLMandelbrotWorker.java*, all this is done in a method named `handleConnection()` that is called to handle a connection that has already been opened to the master program. It uses a method `readTask()` to decode a task that it receives from the master and a method `writeResults()` to encode the results of the task for transmission back to the master. It must also handle any errors that occur:

```
private static void handleConnection(Socket connection) {
   try {
      BufferedReader in = new BufferedReader(
                new InputStreamReader( connection.getInputStream()) );
      PrintWriter out = new PrintWriter(connection.getOutputStream());
      while (true) {
         String line = in.readLine();  // Message from the master.
         if (line == null) {
               // End-of-stream encountered -- should not happen.
```

```
                throw new Exception("Connection closed unexpectedly.");
             }
             if (line.startsWith(CLOSE_CONNECTION_COMMAND)) {
                   // Represents the normal termination of the connection.
                System.out.println("Received close command.");
                break;
             }
             else if (line.startsWith(TASK_COMMAND)) {
                   // Represents a CLMandelbrotTask that this worker is
                   // supposed to perform.
                CLMandelbrotTask task = readTask(line);  // Decode the message.
                task.compute();  // Perform the task.
                out.println(writeResults(task));  //  Send back the results.
                out.flush();  // Make sure data is sent promptly!
             }
             else {
                   // No other messages are part of the protocol.
                throw new Exception("Illegal command received.");
             }
          }
       }
       catch (Exception e) {
          System.out.println("Client connection closed with error " + e);
       }
       finally {
          try {
             connection.close();  // Make sure the socket is closed.
          }
          catch (Exception e) {
          }
       }
    }
```

Note that this method is **not** executed in a separate thread. The worker has only one thing to do at a time and does not need to be multithreaded.

Turning to the master program, *CLMandelbrotMaster.java*, we encounter a more complex situation. The master program must communicate with several workers over several network connections. To accomplish this, the master program is multi-threaded, with one thread to manage communication with each worker. A pseudocode outline of the `main()` routine is quite simple:

```
create a list of all tasks that must be performed
if there are no command line arguments {
      // The master program does all the tasks itself.
   Perform each task.
}
else {
      // The tasks will be performed by worker programs.
   for each command line argument:
      Get information about a worker from command line argument.
      Create and start a thread to communicate with the worker.
   Wait for all threads to terminate.
}
// All tasks are now complete (assuming no error occurred).
```

The list of tasks is stored in a variable, `tasks`, of type *ConcurrentBlockingQueue<CLMandelbrotTask>*, `tasks`. (See Subsection 12.3.2.) The communication threads take tasks from this list and send them to worker programs. The method `tasks.poll()` is used to remove a task from the queue. If the queue is empty, it returns `null`, which acts as a signal that all tasks have been assigned and the communication thread can terminate.

The job of a thread is to send a sequence of tasks to a worker thread and to receive the results that the worker sends back. The thread is also responsible for opening the connection in the first place. A pseudocode outline for the process executed by the thread might look like:

```
Create a socket connected to the worker program.
Create input and output streams for communicating with the worker.
while (true) {
   Let task = tasks.poll().
   If task == null
      break;  // All tasks have been assigned.
   Encode the task into a message and transmit it to the worker.
   Read the response from the worker.
   Decode and process the response.
}
Send a "close" command to the worker.
Close the socket.
```

This would work OK. However, there are a few subtle points. First of all, the thread must be ready to deal with a network error. For example, a worker might shut down unexpectedly. But if that happens, the master program can continue, provided other workers are still available. (You can try this when you run the program: Stop one of the worker programs, with `CONTROL-C`, and observe that the master program still completes successfully.) A difficulty arises if an error occurs while the thread is working on a task: If the problem as a whole is going to be completed, that task will have to be reassigned to another worker. I take care of this by putting the uncompleted task back into the task list. (Unfortunately, my program does not handle all possible errors. If the last worker thread fails, there will be no one left to take over the uncompleted task. Also, if a network connection "hangs" indefinitely without actually generating an error, my program will also hang, waiting for a response from a worker that will never arrive. A more robust program would have some way of detecting the problem and reassigning the task.)

Another defect in the procedure outlined above is that it leaves the worker program idle while the thread is processing the worker's response. It would be nice to get a new task to the worker before processing the response from the previous task. This would keep the worker busy and allow two operations to proceed simultaneously instead of sequentially. (In this example, the time it takes to process a response is so short that keeping the worker waiting while it is done probably makes no significant difference. But as a general principle, it's desirable to have as much parallelism as possible in the algorithm.) We can modify the procedure to take this into account:

```
try {
   Create a socket connected to the worker program.
   Create input and output streams for communicating with the worker.
   Let currentTask = tasks.poll().
   Encode currentTask into a message and send it to the worker.
   while (true) {
      Read the response from the worker.
```

```
      Let nextTask = tasks.poll().
      If nextTask != null {
            // Send nextTask to the worker before processing the
            // response to currentTask.
         Encode nextTask into a message and send it to the worker.
      }
      Decode and process the response to currentTask.
      currentTask = nextTask.
      if (currentTask == null)
         break; // All tasks have been assigned.
   }
   Send a "close" command to the worker.
   Close the socket.
}
catch (Exception e) {
   Put uncompleted task, if any, back into the task queue.
}
finally {
   Close the connection.
}
```

Finally, here is how this translates into Java. The pseudocode presented above becomes the `run()` method in the class that defines the communication threads used by the master program:

```
/**
 * This class represents one worker thread.  The job of a worker thread
 * is to send out tasks to a CLMandelbrotWorker program over a network
 * connection, and to get back the results computed by that program.
 */
private static class WorkerConnection extends Thread {

    int id;        // Identifies this thread in output statements.
    String host;   // The host to which this thread will connect.
    int port;      // The port number to which this thread will connect.

    /**
     * The constructor just sets the values of the instance
     * variables id, host, and port and starts the thread.
     */
    WorkerConnection(int id, String host, int port) {
        this.id = id;
        this.host = host;
        this.port = port;
        start();
    }

    /**
     * The run() method of the thread opens a connection to the host and
     * port specified in the constructor, then sends tasks to the
     * CLMandelbrotWorker program on the other side of that connection.
     * If the thread terminates normally, it outputs  the number of tasks
     * that it processed.  If it terminates with an error, it outputs
     * an error message.
     */
    public void run() {
```

```
int tasksCompleted = 0; // How many tasks has this thread handled.
Socket socket;  // The socket for the connection.

try {
    socket = new Socket(host,port);  // open the connection.
}
catch (Exception e) {
    System.out.println("Thread " + id +
            " could not open connection to " + host + ":" + port);
    System.out.println("   Error: " + e);
    return;
}

CLMandelbrotTask currentTask = null;
CLMandelbrotTask nextTask = null;

try {
    PrintWriter out = new PrintWriter(socket.getOutputStream());
    BufferedReader in = new BufferedReader(
                      new InputStreamReader(socket.getInputStream()) );
    currentTask = tasks.poll();
    if (currentTask != null) {
           // Send first task to the worker program.
        String taskString = writeTask(currentTask);
        out.println(taskString);
        out.flush();
    }
    while (currentTask != null) {
        String resultString = in.readLine(); // Get results for currentTask.
        if (resultString == null)
            throw new IOException("Connection closed unexpectedly.");
        if (! resultString.startsWith(RESULT_COMMAND))
            throw new IOException("Illegal string received from worker.");
        nextTask = tasks.poll();  // Get next task and send it to worker.
        if (nextTask != null) {
               // Send nextTask to worker before processing results for
               // currentTask, so that the worker can work on nextTask
               // while the currentTask results are processed.
            String taskString = writeTask(nextTask);
            out.println(taskString);
            out.flush();
        }
        readResults(resultString, currentTask);
        finishTask(currentTask);  // Process results from currentTask.
        tasksCompleted++;
        currentTask = nextTask;   // We are finished with old currentTask.
        nextTask = null;
    }
    out.println(CLOSE_CONNECTION_COMMAND);  // Send close command to worker.
    out.flush();
}
catch (Exception e) {
    System.out.println("Thread " + id + " terminated because of an error");
    System.out.println("   Error: " + e);
    e.printStackTrace();
```

```
                // Put uncompleted tasks, if any, back into the task list.
            if (currentTask != null)
                reassignTask(currentTask);
            if (nextTask != null)
                reassignTask(nextTask);
        }
        finally {
            System.out.println("Thread " + id + " ending after completing " +
                    tasksCompleted + " tasks");
            try {
                socket.close();
            }
            catch (Exception e) {
            }
        }

    } //end run()

} // end nested class WorkerConnection
```

12.5 Network Programming Example: A Networked Game Framework

This section presents several programs that use networking and threads. The common problem in each application is to support network communication between several programs running on different computers. A typical example of such an application is a networked game with two or more players, but the same problem can come up in less frivolous applications as well. The first part of this section describes a common framework that can be used for a variety of such applications, and the rest of the section discusses three specific applications that use that framework.

This section was inspired by a pair of students, Alexander Kittelberger and Kieran Koehnlein, who wanted to write a networked poker game as a final project in a class that I was teaching. I helped them with the network part of the project by writing a basic framework to support communication between the players. Since the application illustrates a variety of important ideas, I decided to include a somewhat more advanced and general version of that framework in the current edition of this book. The final example is a networked poker game.

12.5.1 The Netgame Framework

One can imagine playing many different games over the network. As far as the network goes, all of those games have at least one thing in common: There has to be some way for actions taken by one player to be communicated over the network to other players. It makes good programming sense to make that capability available in a reusable common core that can be used in many different games. I have written such a core; it is defined by several classes in the package *netgame.common*.

We have not done much with packages in this book, aside from using built-in classes. Packages were introduced in Subsection 2.6.4, but we have stuck to the "default package" in our programming examples. In practice, however, packages are used in all but the simplest programming projects to divide the code into groups of related classes. It makes particularly good

sense to define a reusable framework in a package that can be included as a unit in a variety of projects.

Integrated development environments such as Eclipse or Netbeans make it very easy to use packages: To use the netgame package in a project in an IDE, simply copy-and-paste the entire *netgame* directory into the project.

If you work on the command line, you should be in a working directory that includes the *netgame* directory as a subdirectory. Then, to compile all the java files in the package *netgame.common*, for example, you can use the following command in Mac OS or Linux:

```
javac netgame/common/*.java
```

For windows, you should use backslashes instead of forward slashes:

```
javac netgame\common\*.java
```

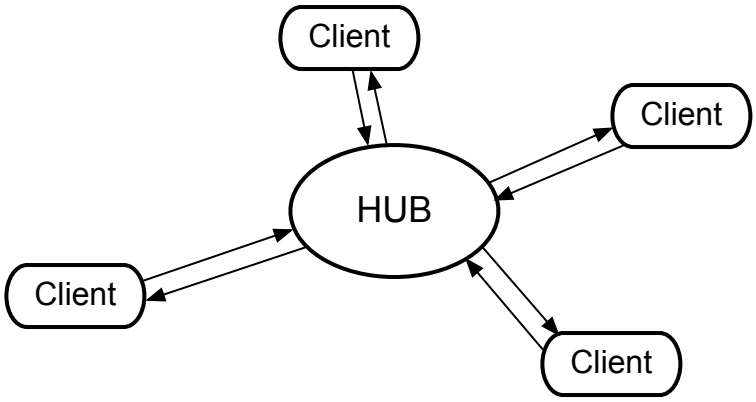
To run a main program that is defined in a package, you should again be in a directory that contains the package as a subdirectory, and you should use the full name of the class that you want to run. For example, the *ChatRoomServer* class, discussed later in this section, is defined in the package *netgame.chat*, so you would run it with the command

```
java netgame.chat.ChatRoomServer
```

I will have more to say about packages in the final example of the book, in Section 13.5.

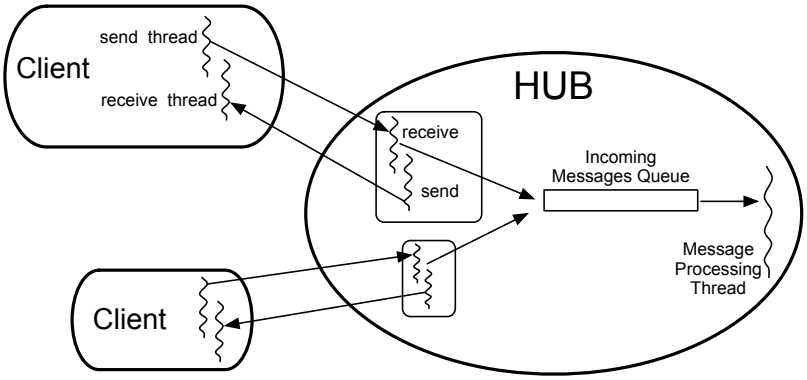
* * *

The applications discussed in this section are examples of distributed computing, since they involve several computers communicating over a network. Like the example in Subsection 12.4.5, they use a central "server," or "master," to which a number of "clients" will connect. All communication goes through the server; a client cannot send messages directly to another client. In this section, I will refer to the server as a ***hub***, in the sense of "communications hub":



In Subsection 12.4.5, messages were sent back and forth between the server and the client in a definite, predetermined sequence. Communication between the server and a client was actually communication between one thread running on the server and another thread running on the client. For the netgame framework, however, I want to allow for asynchronous communication, in which it is not possible to wait for messages to arrive in a predictable sequence. To make this possible a netgame client will use two threads for communication, one for sending messages and one for receiving messages. Similarly, the netgame hub will use two threads for communicating with each client.

The hub is generally connected to many clients and can receive messages from any of those clients at any time. The hub will have to process each message in some way. To organize this processing, the hub uses a single thread to process all incoming messages. When a communication thread receives a message from a client, it simply drops that message into a queue of incoming messages. There is only one such queue, which is used for messages from all clients. The message processing thread runs in a loop in which it removes a message from the queue, processes it, removes another message from the queue, processes it, and so on. The queue itself is implemented as an object of type *LinkedBlockingQueue* (see Subsection 12.3.3).



There is one more thread in the hub, not shown in the illustration. This final thread creates a *ServerSocket* and uses it to listen for connection requests from clients. Each time it accepts a connection request, it hands off the client to another object, defined by the nested class *ConnectionToClient*, which will handle communication with that client. Each connected client is identified by an ID number. ID numbers 1, 2, 3, ... are assigned to clients as they connect. Since clients can also disconnect, the clients connected at any give time might not have consecutive IDs. A variable of type *TreeMap<Integer,ConnectionToClient>* associates the ID numbers of connected clients with the objects that handle their connections.

The messages that are sent and received are objects. The I/O streams that are used for reading and writing objects are of type *ObjectInputStream* and *ObjectOutputStream*. (See Subsection 11.1.6.) The output stream of a socket is wrapped in an *ObjectOutputStream* to make it possible to transmit objects through that socket. The socket's input stream is wrapped in an *ObjectInputStream* to make it possible to receive objects. Remember that the objects that are used with such streams must implement the interface `java.io.Serializable`.

The netgame *Hub* class is defined in the file *Hub.java*, in the package *netgame.common*. The port on which the server socket will listen must be specified as a parameter to the *Hub* constructor. The *Hub* class defines a method

```
protected void messageReceived(int playerID, Object message)
```

which is called to process messages that are received from clients. The first parameter, `playerID`, is the ID number of the client from whom the message was received, and the second parameter is the message itself. In the *Hub* class, this method will simply forward the message to all connected clients. To forward the message, it wraps both the `playerID` and the `message` in an object of type *ForwardedMessage* (defined in the file *ForwardedMessage.java*, in the package `netgame.common`). In a simple application such as the chat room discussed in the next subsection, this might be sufficient. For most applications, however, it will be necessary to

define a subclass of *Hub* and redefine the `messageReceived()` method to do more complicated message processing. There are several other methods that a subclass might redefine, including

- `protected void playerConnected(int playerID)` — This method is called each time a player connects to the hub. The parameter `playerID` is the ID number of the newly connected player. In the *Hub* class, this method does nothing. Note that the complete list of ID numbers for currently connected players can be obtained by calling `getPlayerList()`.
- `protected void playerDisconnected(int playerID)` — This is called each time a player disconnects from the hub. The parameter tells which player has just disconnected. In the *Hub* class, this method does nothing.

The *Hub* class also defines a number of useful public methods, notably

- `sendToAll(message)` — sends the specified `message` to every client that is currently connected to the hub. The message must be a non-null object that implements the *Serializable* interface.
- `sendToOne(recipientID,message)` — sends a specified `message` to just one user. The first parameter, `recipientID` is the ID number of the client who will receive the message. This method returns a **boolean** value, which is false if there is no connected client with the specified `recipientID`.
- `shutDownServerSocket()` — shuts down the hub's server socket, so that no additional clients will be able to connect. This could be used, for example, in a two-person game, after the second client has connected.
- `setAutoreset(autoreset)` — sets the **boolean** value of the `autoreset` property. If this property is `true`, then the *ObjectOutputStreams* that are used to transmit messages to clients will automatically be reset before each message is transmitted. (Resetting an *ObjectOutputStream* is something that has to be done if an object is written to the stream, modified, and then written to the stream again. If the stream is not reset before writing the modified object, then the old, unmodified value is sent to the stream instead of the new value. See Subsection 11.1.6 for a discussion of this technicality.)

For more information—and to see how all this is implemented—you should read the source code file *Hub.java*. With some effort, you should be able to understand everything in that file.

* * *

Turning to the client side, the basic netgame client class is defined in the file *Client.java*, in the package *netgame.common*. The *Client* class has a constructor that specifies the host name (or IP address) and port number of the hub to which the client will connect. This constructor blocks until the connection has been established.

Client is an `abstract` class. Every netgame application must define a subclass of *Client* and provide a definition for the abstract method:

```
abstract protected void messageReceived(Object message);
```

This method is called each time a message is received from the netgame hub to which the client is connected. A subclass of client might also override the `protected` methods `playerConnected`, `playerDisconnected`, `serverShutdown`, and `connectionClosedByError`. See the *source code* for more information. I should also note that *Client* contains the `protected` instance variable `connectedPlayerIDs`, of type `int[]`, an array containing the ID numbers of all the clients that are currently connected to the hub. The most important `public` methods that are provided by the `Client` class are

- `send(message)` — transmits a message to the hub. The `message` can be any non-null object that implements the *Serializable* interface.
- `getID()` — gets the ID number that was assigned to this client by the hub.
- `disconnect()` — closes the client's connection to the hub. It is not possible to send messages after disconnecting. The `send()` method will throw an *IllegalStateException* if an attempt is made to do so.

The *Hub* and *Client* classes are meant to define a general framework that can be used as the basis for a variety of networked games—and, indeed, of other distributed programs. The low level details of network communication and multithreading are hidden in the `private` sections of these classes. Applications that build on these classes can work in terms of higher-level concepts such as players and messages. The design of these classes was developed though several iterations, based on experience with several actual applications. I urge you to look at the source code to see how *Hub* and *Client* use threads, sockets, and streams. In the remainder of this section, I will discuss three applications built on the netgame framework. I will not discuss these applications in great detail. You can find the complete source code for all three in the *netgame* package.

12.5.2 A Simple Chat Room

Our first example is a "chat room," a network application where users can connect to a server and can then post messages that will be seen by all current users of the room. It is similar to the *GUIChat* program from Subsection 12.4.2, except that any number of users can participate in a chat. While this application is not a game as such, it does show the basic functionality of the netgame framework.

The chat room application consists of two programs. The first, *ChatRoomServer.java*, is a completely trivial program that simply creates a netgame *Hub* to listen for connection requests from netgame clients:

```
public static void main(String[] args) {
    try {
        new Hub(PORT);
    }
    catch (IOException e) {
        System.out.println("Can't create listening socket.  Shutting down.");
    }
}
```

The port number, `PORT`, is defined as a constant in the program and is arbitrary, as long as both the server and the clients use the same port.

The second part of the chat room application is the program *ChatRoomWindow.java*, which is meant to be run by users who want to participate in the chat room. A potential user must know the name (or IP address) of the computer where the hub is running. (For testing, it is possible to run the client program on the same computer as the hub, using `localhost` as the name of the computer where the hub is running.) When *ChatRoomWindow* is executed, it uses a dialog box to ask the user for this information. It then opens a window that will serve as the user's interface to the chat room. The window has a large transcript area that displays messages that users post to the chat room. It also has a text input box where the user can enter messages. When the user enters a message, that message will be posted to the transcript

of every user who is connected to the hub, so all users see every message sent by every user. Let's look at some of the programming.

Any netgame application must define a subclass of the abstract *Client* class. For the chat room application, clients are defined by a nested class *ChatClient* inside *ChatRoomWindow*. The program has an instance variable, `connection`, of type *ChatClient*, which represents the program's connection to the hub. When the user enters a message, that message is sent to the hub by calling

```
connection.send(message);
```

When the hub receives the message, it packages it into an object of type *ForwardedMessage*, along with the ID number of the client who sent the message. The hub sends a copy of that *ForwardedMessage* to every connected client, including the client who sent the message. When the message is received from the hub by a client object, the `messageReceived()` method of the client object is called. *ChatClient* overrides this method to make it add the message to the transcript of the *ChatClientWindow*.

A client is also notified when a player connects to or disconnects from the hub and when the connection with the hub is lost. *ChatClient* overrides the methods that are called when these events happen so that they post appropriate messages to the transcript. Here's the complete definition of the client class for the chat room application:

```
/**
 * A ChatClient connects to a Hub and is used to send messages to
 * and receive messages from the Hub.  Messages received from the
 * Hub will be of type ForwardedMessage and will contain the
 * ID number of the sender and the string that was sent by that user.
 */
private class ChatClient extends Client {

    /**
     * Opens a connection to the chat room server on a specified computer.
     */
    ChatClient(String host) throws IOException {
        super(host, PORT);
    }

    /**
     * Responds when a message is received from the server.  It should be
     * a ForwardedMessage representing something that one of the participants
     * in the chat room is saying.  The message is simply added to the
     * transcript, along with the ID number of the sender.
     */
    protected void messageReceived(Object message) {
        if (message instanceof ForwardedMessage) {
                        // (no other message types are expected)
            ForwardedMessage fm = (ForwardedMessage)message;
            addToTranscript("#" + fm.senderID + " SAYS:  " + fm.message);
        }
    }

    /**
     * Called when the connection to the client is shut down because of some
     * error message.  (This will happen if the server program is terminated.)
     */
```

```
        protected void connectionClosedByError(String message) {
            addToTranscript("Sorry, communication has shut down due to an error:\n     "
                                            + message);
            sendButton.setEnabled(false);
            messageInput.setEnabled(false);
            messageInput.setEditable(false);
            messageInput.setText("");
            connected = false;
            connection = null;
        }

        /**
         * Posts a message to the transcript when someone leaves the chat room.
         */
        protected void playerConnected(int newPlayerID) {
            addToTranscript("Someone new has joined the chat room, with ID number "
                                           + newPlayerID);
        }

        /**
         * Posts a message to the transcript when someone leaves the chat room.
         */
        protected void playerDisconnected(int departingPlayerID) {
            addToTranscript("The person with ID number " + departingPlayerID
                                         + " has left the chat room");
        }

    } // end nested class ChatClient
```

For the full source code of the chat room application, see the source code files, which can be found in the package *netgame.chat*.

Note: A user of my chat room application is identified only by an ID number that is assigned by the hub when the client connects. Essentially, users are anonymous, which is not very satisfying. See Exercise 12.6 at the end of this chapter for a way of addressing this issue.

12.5.3 A Networked TicTacToe Game

My second example is a very simple game: the familiar children's game TicTacToe. In TicTacToe, two players alternate placing marks on a three-by-three board. One player plays X's; the other plays O's. The object is to get three X's or three O's in a row.

At a given time, the state of a TicTacToe game consists of various pieces of information such as the current contents of the board, whose turn it is, and—when the game is over—who won or lost. In a typical non-networked version of the game, this state would be represented by instance variables. The program would consult those instance variables to determine how to draw the board and how to respond to user actions such as mouse clicks. In the networked netgame version, however, there are **three** programs involved: Two copies of a client program, which provide the interface to the two players of the game, and the hub program that manages the connections to the clients. These programs are not even running on the same computer, so they can't share the same instance variables. Nevertheless, the game has to have a single, well-defined state at any time, and both players have to be aware of that state.

My solution is to store the "official" game state in the hub, and to send a copy of that state to each player every time the state changes. The players can't change the state directly. When a player takes some action, such as placing a piece on the board, that action is sent as a message

to the hub. The hub changes the state to reflect the result of the action, and it sends the new state to both players. The window used by each player will then be updated to reflect the new state. In this way, we can be sure that the game always looks the same to both players.

Networked TicTacToe is defined in several classes in the package *netgame.tictactoe*. *TicTacToeGameState* represents the state of a game. It includes a method

```
public void applyMessage(int senderID, Object message)
```

that modifies the state to reflect the effect of a message received from one of the players of the game. The message will represent some action taken by the player, such as clicking on the board.

The *Hub* class knows nothing about TicTacToe. Since the hub for the TicTacToe game has to keep track of the state of the game, it has to be defined by a subclass of *Hub*. The *TicTacToeGameHub* class is quite simple. It overrides the `messageReceived()` method so that it responds to a message from a player by applying that message to the game state and sending a copy of the new state to both players. It also overrides the `playerConnected()` and `playerDisconnected()` methods to take appropriate actions, since the game can only be played when there are exactly two connected players. Here is the complete source code:

```
package netgame.tictactoe;

import java.io.IOException;

import netgame.common.Hub;

/**
 * A "Hub" for the network TicTacToe game.  There is only one Hub
 * for a game, and both network players connect to the same Hub.
 * Official information about the state of the game is maintained
 * on the Hub.  When the state changes, the Hub sends the new
 * state to both players, ensuring that both players see the
 * same state.
 */
public class TicTacToeGameHub extends Hub {

    private TicTacToeGameState state;  // Records the state of the game.

    /**
     * Create a hub, listening on the specified port.  Note that this
     * method calls setAutoreset(true), which will cause the output stream
     * to each client to be reset before sending each message.  This is
     * essential since the same state object will be transmitted over and
     * over, with changes between each transmission.
     * @param port the port number on which the hub will listen.
     * @throws IOException if a listener cannot be opened on the specified port.
     */
    public TicTacToeGameHub(int port) throws IOException {
        super(port);
        state = new TicTacToeGameState();
        setAutoreset(true);
    }

    /**
     * Responds when a message is received from a client.  In this case,
     * the message is applied to the game state, by calling state.applyMessage().
     * Then the possibly changed state is transmitted to all connected players.
```

```
         */
        protected void messageReceived(int playerID, Object message) {
            state.applyMessage(playerID, message);
            sendToAll(state);
        }

        /**
         * This method is called when a player connects.  If that player
         * is the second player, then the server's listening socket is
         * shut down (because only two players are allowed), the
         * first game is started, and the new state -- with the game
         * now in progress -- is transmitted to both players.
         */
        protected void playerConnected(int playerID) {
            if (getPlayerList().length == 2) {
                shutdownServerSocket();
                state.startFirstGame();
                sendToAll(state);
            }
        }

        /**
         * This method is called when a player disconnects.  This will
         * end the game and cause the other player to shut down as
         * well.  This is accomplished by setting state.playerDisconnected
         * to true and sending the new state to the remaining player, if
         * there is one, to notify that player that the game is over.
         */
        protected void playerDisconnected(int playerID) {
            state.playerDisconnected = true;
            sendToAll(state);
        }
    }
```

A player's interface to the game is represented by the class *TicTacToeWindow*. As in the chat room application, this class defines a nested subclass of *Client* to represent the client's connection to the hub. One interesting point is how the client responds to a message from the hub. Such a message represents a new game state. When the message is received, the window must be updated to show the new state. The message is received and processed by one thread; the updating is done in another thread. This has the potential of introducing race conditions that require synchronization. (In particular, as I was developing the program, I found that it was possible for a message to be received before the window's constructor had finished executing. This led to a very hard-to-diagnose bug because my response to the message was trying to use objects that had not yet been created.)

When working with the Swing API, it is recommended that all modifications to the GUI be made in the GUI event thread. An alternative would be to make `paintComponent()` and other methods `synchronized`, but that would negatively impact the performance of the GUI. Swing includes a method `SwingUtilitites.invokeLater(runnable)` to make it possible to run arbitrary code in the GUI event thread. The parameter, `runnable`, is an object that implements the *Runnable* interface that was discussed in Subsection 12.1.1. A *Runnable* object has a `run()` method. `SwingUtilities.runLater()` will schedule the `run()` method of the object to be executed in the GUI event thread. It will be executed after that thread has

finished handling any pending events. By executing `run()` in the event thread, you can be sure that it will not introduce any synchronization problems. In the TicTacToe client class, this technique is used in the method that processes events received from the hub:

```
protected void messageReceived(final Object message) {
    if (message instanceof TicTacToeGameState) {
        SwingUtilities.invokeLater(new Runnable(){
            public void run() {
                   // The newState() method updates the GUI for the new state.
                newState( (TicTacToeGameState)message );
            }
        });
    }
}
```

(The *SwingUtilities* class, by the way, includes a variety of useful static methods that can be used in programming with Swing; it's worth taking a look at the documentation for that class.)

To run the TicTacToe netgame, the two players should each run the program *Main.java* in the package *netgame.tictactoe*. This program presents the user with a dialog box where the user can choose to start a new game or to join an existing game. If the user starts a new game, then a *TicTacToeHub* is created to manage the game; a *TicTacToeWindow* is created and connects to that hub. If the user chooses to connect to an existing game, then only the window is created; that window connects to the hub that was created by the first player. The second player has to know the name of the computer where the first player's program is running. As usual, for testing, you can run everything on one computer and use "localhost" as the computer name.

12.5.4 A Networked Poker Game

And finally, we turn very briefly to the application that inspired the netgame framework: Poker. In particular, I have implemented a two-player version of the traditional "five card draw" version of that game. This is a rather complex application, and I do not intend to say much about it here other than to describe the general design. The full source code can be found in the package *netgame.fivecarddraw*. To fully understand it, you will need to be familiar with the game of five card draw poker. And it uses some techniques from Section 13.1 for drawing the cards.

In general outline, the Poker game is similar to the TicTacToe game. There is a *Main* class that can be run by either player, to start a new game or to join an existing game. There is a class *PokerGameState* to represent the state of a game. And there is a subclass, *PokerHub*, of *Hub* to manage the game.

But Poker is a much more complicated game than TicTacToe, and the game state is correspondingly more complicated. It's not clear that we want to broadcast a new copy of the complete game state to the players every time some minor change is made in the state. Furthermore, it doesn't really make sense for both players to know the full game state—that would include the opponent's hand and full knowledge of the deck from which the cards are dealt. (Of course, our client programs wouldn't have to show the full state to the players, but it would be easy enough for a player to substitute their own client program to enable cheating.) So in the Poker application, the full game state is known only to the *PokerHub*. A *PokerGameState* object represents a view of the game from the point of view of one player only. When the state of the game changes, the *PokerHub* creates two different *PokerGameState* objects, representing the state of the game from each player's point of view, and it sends the appropriate game state objects to each player. You can see the *source code* for details.

(One of the hard parts in poker is to implement some way to compare two hands, to see which is higher. In my game, this is handled by the class *PokerRank*. You might find this class useful in other poker games.)

Exercises for Chapter 12

1. Subsection 12.1.3 discusses the need for synchronization in multithreaded programs, and it defines a *ThreadSafeCounter* class with the necessary synchronization. Is this really important? Can you really get errors by using an unsynchronized counter with multiple threads? Write a program to find out. Use the following unsynchronized counter class, which you can include as a nested class in your program:

```
static class Counter {
    int count;
    void inc() {
        count = count+1;
    }
    int getCount() {
        return count;
    }
}
```

Write a thread class that will call the `inc()` method in this class a specified number of times. Create several threads, start them all, and wait for all the threads to terminate. Print the final value of the counter, and see whether it is correct.

Let the user enter the number of threads and the number of times that each thread will increment the counter. You might need a fairly large number of increments to see an error. And of course there can never be any error if you use just one thread. Your program can use `join()` to wait for a thread to terminate (see Subsection 12.1.2).

2. Exercise 3.2 asked you to find the integer in the range 1 to 10000 that has the largest number of divisors. Now write a program that uses multiple threads to solve the same problem, but for the range 1 to 100000 (or less, if you don't have a fast computer). By using threads, your program will take less time to do the computation when it is run on a multiprocessor computer. At the end of the program, output the elapsed time, the integer that has the largest number of divisors, and the number of divisors that it has. The program can be modeled on the sample prime-counting program *ThreadTest2.java* from Subsection 12.1.3. For this exercise, you should simply divide up the problem into parts and create one thread to do each part.

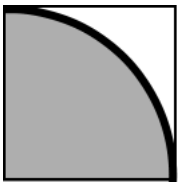
3. In the previous exercise, you divided up a large task into a small number of large pieces and created a thread to execute each task. Because of the nature of the problem, this meant that some threads had much more work to do than others—it is much easier to find the number of divisors of a small number than it is of a big number. As discussed in Subsection 12.3.1, a better approach is to break up the problem into a fairly large number of smaller problems. Subsection 12.3.2 shows how to use a thread pool to execute the tasks: Each thread in the pool runs in a loop in which it repeatedly takes a task from a queue and carries out that task. Implement a thread pool strategy for solving the same maximum-number-of-divisors problem as in the previous exercise.

To make things even more interesting, you should try a new technique for combining the results from all the tasks: Use two queues in your program. Use a queue of tasks, as usual, to hold the tasks that will be executed by the thread pool (Subsection 12.3.2). But also use a queue of results produced by the threads. When a task completes, the result

from that task should be placed into the result queue. The main program can read results from the second queue as they become available, and combine all the results to get the final answer. The result queue will have to be a blocking queue (Subsection 12.3.3), since the main program will have to wait for tasks to become available. Note that the main program knows the exact number of results that it expects to read from the queue, so it can do so in a `for` loop; when the `for` loop completes, the main program knows that all the tasks have been executed.

4. In Exercise 11.3, you wrote a network server program that can send text files from a specified directory to clients. That program used a single thread, which handled all the communication with each client. Modify the program to turn it into a multithreaded server. Use a thread pool of connection-handling threads and use an *ArrayBlockingQueue* to get connected sockets from the `main()` routine to the threads. The sample program *DateServerWithThreads.java* from Subsection 12.4.3 is an example of a multithreaded server that works in this way. Your server program will work with the same client program as the original server. You wrote the client program as the solution to Exercise 11.4.

5. It is possible to get an estimate of the mathematical constant π by using a random process. The idea is based on the fact that the area of a circle of radius 1 is equal to π, and the area of a *quarter* of that circle is $\pi/4$. Here is a picture of a quarter of a circle of radius 1, inside a 1-by-1 square:



The area of the whole square is one, while the area of the part inside the circle is $\pi/4$. If we choose a point in the square at random, the probability that it is inside the circle is $\pi/4$. If we choose `N` points in the square at random, and if `C` of them are inside the circle, we expect the fraction `C/N` of points that fall inside the circle to be about $\pi/4$. That is, we expect `4*C/N` to be close to π. If `N` is large, we can expect `4*C/N` to be a good estimate for π, and as `N` gets larger and larger, the estimate is likely to improve.

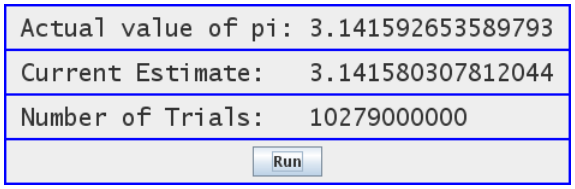
We can pick a random number in the square by choosing numbers `x` and `y` in the range 0 to 1 (using `Math.random()`). Since the equation of the circle is `x*x+y*y=1`, the point lies inside the circle if `x*x+y*y` is less than 1. One trial consists of picking `x` and `y` and testing whether `x*x+y*y` is less than 1. To get an estimate for π, you have to do many trials, count the trials, and count the number of trials in which `x*x+y*y` is less than 1,

For this exercise, you should write a GUI program that does this computation and displays the result. The computation should be done in a separate thread, and the results should be displayed periodically. The program can use *JLabels* to the display the results. It should set the text on the labels after running each batch of, say, one million trials. (Setting the text after each trial doesn't make sense, since millions of trials can be done in one second, and trying to change the display millions of times per second would be silly.

Your program should have a "Run"/"Pause" button that controls the computation. When the program starts, clicking "Run" will start the computation and change the text on the button to "Pause". Clicking "Pause" will cause the computation to pause. The

thread that does the computation should be started at the beginning of the program, but should immediately go into the paused state until the "Run" button is pressed. Use the `wait()` method in the thread to make it wait until "Run" is pressed. Use the `notify()` method when the "Run" button is pressed to wake up the thread. Use a **boolean** signal variable `running` to control whether the computation thread is paused. (The `wait()` and `notify()` methods are covered in Subsection 12.3.4.)

Here is a picture of the program after it has run more than ten billion trials:



You might want to start with a version of the program with no control button. In that version, the computation thread can run continually from the time it is started. Once that is working, you can add the button and the control feature.

To get you started, here is the code from the thread in my solution that runs one batch of trials and updates the display labels:

```
for (int i = 0; i < BATCH_SIZE; i++) {
    double x = Math.random();
    double y = Math.random();
    trialCount++;
    if (x*x + y*y < 1)
        inCircleCount++;
}
double estimateForPi = 4 * ((double)inCircleCount / trialCount);
countLabel.setText(      " Number of Trials:   " + trialCount);
piEstimateLabel.setText( " Current Estimate:   " + estimateForPi);
```

The variables `trialCount` and `inCircleCount` are of type **long** in order to allow the number of trials to be more than the two billion or so that would be possible with a variable of type **int**.

(I was going to ask you to use multiple computation threads, one for each available processor, but I ran into an issue when using the `Math.random()` method in several threads. This method requires synchronization, which causes serious performance problems when several threads are using it to generate large amounts of random numbers. A solution to this problem is to have each thread use its own object of type `java.util.Random` to generate its random numbers (see Subsection 5.3.1). My on-line solution to this exercise discusses this problem further.)

6. The chat room example from Subsection 12.5.2 can be improved in several ways. First, it would be nice if the participants in the chat room could be identified by name instead of by number. Second, it would be nice if one person could send a private message to another person that would be seen just by that person rather than by everyone. Make these two changes. You can start with a copy of the package *netgame.chat*. You will also need the package *netgame.common*, which defines the netgame framework.

To make the first change, you will have to implement a subclass of *Hub* that can keep track of client names as well as numbers. To get the name of a client to the hub, you

can override the `extraHandshake()` method both in the *Hub* subclass and in the *Client* subclass. The `extraHandshake()` method is called as part of setting up the connection between the client and the hub. It is called after the client has been assigned an ID number but before the connection is considered to be fully connected. It should throw an *IOException* if some error occurs during the setup process. Note that messages that are sent by the hub should be read by the client and vice versa. The `extraHandshake()` method in the Client is defined as:

```
protected void extraHandshake(ObjectInputStream in, ObjectOutputStream out)
                                                        throws IOException
```

while in the Hub, there is an extra parameter that tells the ID number of the client whose connection is being set up:

```
protected void extraHandshake(in playerID, ObjectInputStream in,
                              ObjectOutputStream out) throws IOException
```

In the *ChatRoomWindow* class, the `main()` routine asks the user for the name of the computer where the server is running. You can add some code there to ask the user their name. You will have to decide what to do if two users want to use the same name. You might consider having a list of users who are allowed to join the chat room. You might even assign them passwords.

For the second improvement, personal messages, I suggest defining a *PrivateMessage* class. A *PrivateMessage* object would include both the string that represents the message and the ID numbers of the player to whom the message is being sent and the player who sent the message. The hub will have to be programmed to know how to deal with such messages. A *PrivateMessage* should only be sent to the client who is listed as the recipient of the message. You need to decide how the user will input a private message and how the user will select the recipient of the message.

Quiz on Chapter 12

1. Write a complete subclass of *Thread* to represent a thread that writes out the numbers from 1 to 10. Then write some code that would create and start a thread belonging to that class.

2. Suppose that `thrd` is an object of type *Thread*. Explain the difference between calling `thrd.start()` and calling `thrd.run()`.

3. What is a *race condition*?

4. How does synchronization prevent race conditions, and what does it mean to say that synchronization only provides *mutual* exclusion?

5. Suppose that a program uses a single thread that takes 4 seconds to run. Now suppose that the program creates two threads and divides the same work between the two threads. What can be said about the expected execution time of the program that uses two threads?

6. What is an *ArrayBlockingQueue* and how does it solve the producer/consumer problem?

7. What is a *thread pool*?

8. Network server programs are often *multithreaded.* Explain what this means and why it is true.

9. Why does a multithreaded network server program often use many times more threads than the number of available processors?

10. Consider the *ThreadSafeCounter* example from Subsection 12.1.3:

```
public class ThreadSafeCounter {

   private int count = 0;  // The value of the counter.

   synchronized public void increment() {
      count = count + 1;
   }

   synchronized public int getValue() {
      return count;
   }

}
```

The `increment()` method is synchronized so that the caller of the method can complete the three steps of the operation "Get value of count," "Add 1 to value," "Store new value in count" without being interrupted by another thread. But `getValue()` consists of a single, simple step. Why is `getValue()` synchronized? (This is a deep and tricky question.)

Chapter 13

Advanced GUI Programming

It's possible to program a wide variety of GUI applications using only the techniques covered in Chapter 6. In many cases, the basic events, components, layouts, and graphics routines covered in that chapter suffice. But the Swing graphical user interface library is far richer than what we have seen so far, and it can be used to build highly sophisticated applications. This chapter is a further introduction to Swing and other aspects of GUI programming. Although the title of the chapter is "Advanced GUI Programming," it is still just an introduction. Full coverage of this topic would require at least another complete book.

13.1 Images and Resources

We have seen how to use the *Graphics* class to draw on a GUI component that is visible on the computer's screen. Often, however, it is useful to be able to create a drawing **off-screen**, in the computer's memory. It is also important to be able to work with images that are stored in files.

To a computer, an image is just a set of numbers. The numbers specify the color of each pixel in the image. The numbers that represent the image on the computer's screen are stored in a part of memory called a ***frame buffer***. Many times each second, the computer's video card reads the data in the frame buffer and colors each pixel on the screen according to that data. Whenever the computer needs to make some change to the screen, it writes some new numbers to the frame buffer, and the change appears on the screen a fraction of a second later, the next time the screen is redrawn by the video card.

Since it's just a set of numbers, the data for an image doesn't have to be stored in a frame buffer. It can be stored elsewhere in the computer's memory. It can be stored in a file on the computer's hard disk. Just like any other data file, an image file can be downloaded over the Internet. Java includes standard classes and subroutines that can be used to copy image data from one part of memory to another and to get data from an image file and use it to display the image on the screen.

13.1.1 Images and BufferedImages

The class `java.awt.Image` represents an image stored in the computer's memory. There are two fundamentally different types of *Image*. One kind represents an image read from a source outside the program, such as from a file on the computer's hard disk or over a network connection. The second type is an image created by the program. I refer to this second type as an ***off-screen canvas***. An off-screen canvas is a region of the computer's memory that can be used as a

drawing surface. It is possible to draw to an off-screen image using the same *Graphics* class that is used for drawing on the screen.

An *Image* of either type can be copied onto the screen (or onto an off-screen canvas) using methods that are defined in the *Graphics* class. This is most commonly done in the `paintComponent()` method of a *JComponent*. Suppose that `g` is the *Graphics* object that is provided as a parameter to the `paintComponent()` method, and that `img` is of type *Image*. Then the statement

```
g.drawImage(img, x, y, this);
```

will draw the image `img` in a rectangular area in the component. The integer-valued parameters `x` and `y` give the position of the upper-left corner of the rectangle in which the image is displayed, and the rectangle is just large enough to hold the image. The fourth parameter, `this`, is the special variable from Subsection 5.6.1 that refers to the *JComponent* itself. This parameter is there for technical reasons having to do with the funny way Java treats image files. For most applications, you don't need to understand this, but here is how it works: `g.drawImage()` does not actually draw the image in all cases. It is possible that the complete image is not available when this method is called; this can happen, for example, if the image has to be read from a file. In that case, `g.drawImage()` merely **initiates** the drawing of the image and returns immediately. Pieces of the image are drawn later, asynchronously, as they become available. The question is, **how** do they get drawn? That's where the fourth parameter to the `drawImage` method comes in. The fourth parameter is something called an *ImageObserver*. When a piece of the image becomes available to be drawn, the system will inform the *ImageObserver*, and that piece of the image will appear on the screen. Any *JComponent* object can act as an *ImageObserver*. The `drawImage` method returns a **boolean** value to indicate whether the image has actually been drawn or not when the method returns. When drawing an image that you have created in the computer's memory, or one that you are sure has already been completely loaded, you can set the *ImageObserver* parameter to `null`. This is true in particular for any *BufferedImage*

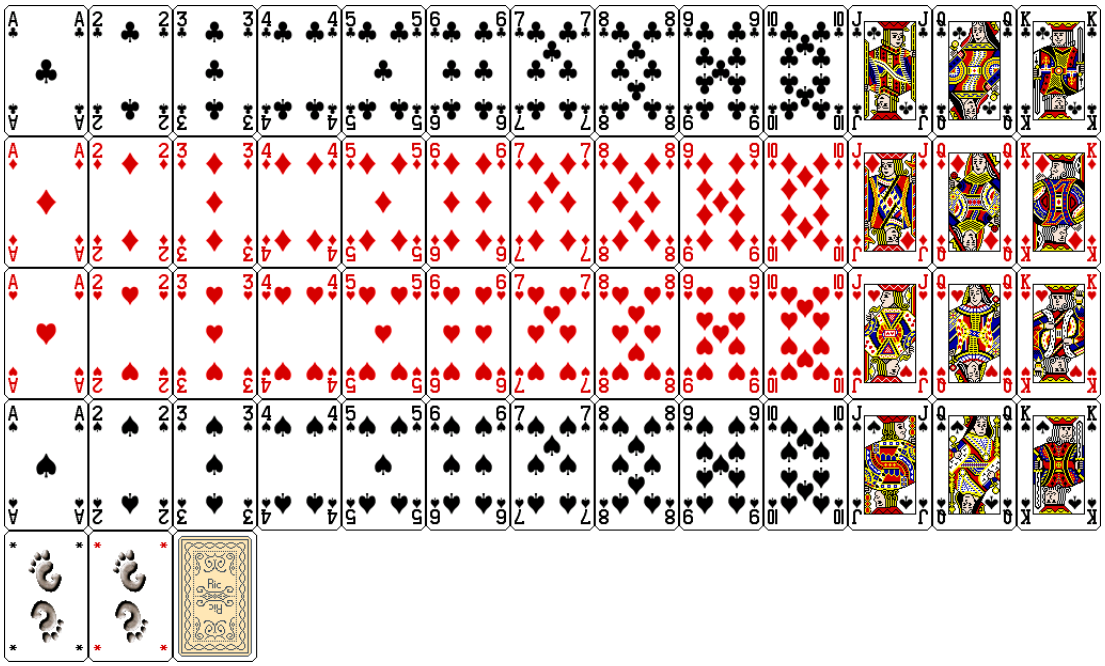
There are a few useful variations of the `drawImage()` method. For example, it is possible to scale the image as it is drawn to a specified width and height. This is done with the command

```
g.drawImage(img, x, y, width, height, imageObserver);
```

The parameters `width` and `height` give the size of the rectangle in which the image is displayed. Another version makes it possible to draw just part of the image. In the command:

```
g.drawImage(img, dest_x1, dest_y1, dest_x2, dest_y2,
                 source_x1, source_y1, source_x2, source_y2, imageObserver);
```

the integers `source_x1`, `source_y1`, `source_x2`, and `source_y2` specify the top-left and bottom-right corners of a rectangular region in the source image. The integers `dest_x1`, `dest_y1`, `dest_x2`, and `dest_y2` specify the corners of a region in the destination graphics context. The specified rectangle in the image is drawn, with scaling if necessary, to the specified rectangle in the graphics context. For an example in which this is useful, consider a card game that needs to display 52 different cards. Dealing with 52 image files can be cumbersome and inefficient, especially for downloading over the Internet. So, all the cards might be put into a single image:



(This image is from the Gnome desktop project, http://www.gnome.org, and is shown here much smaller than its actual size.) Now just one *Image* object is needed. Drawing one card means drawing a rectangular region from the image. This technique is used in a variation of the sample program *HighLowGUI.java* from Subsection 6.7.6. In the original version, the cards are represented by textual descriptions such as "King of Hearts." In the new version, *HighLowWithImages.java*, the cards are shown as images. An applet version of the program can be found in the on-line version of this section.

In the program, the cards are drawn using the following method. The instance variable `cardImages` is a variable of type *Image* that represents the image that is shown above, containing 52 cards, plus two Jokers and a face-down card. Each card is 79 by 123 pixels. These numbers are used, together with the suit and value of the card, to compute the corners of the source rectangle for the `drawImage()` command:

```
/**
 * Draws a card in a 79x123 pixel rectangle with its
 * upper left corner at a specified point (x,y).  Drawing the card
 * requires the image file "cards.png".
 * @param g The graphics context used for drawing the card.
 * @param card The card that is to be drawn.  If the value is null, then a
 * face-down card is drawn.
 * @param x the x-coord of the upper left corner of the card
 * @param y the y-coord of the upper left corner of the card
 */
public void drawCard(Graphics g, Card card, int x, int y) {
   int cx;    // x-coord of upper left corner of the card inside cardsImage
   int cy;    // y-coord of upper left corner of the card inside cardsImage
   if (card == null) {
      cy = 4*123;   // coords for a face-down card.
      cx = 2*79;
   }
   else {
```

```
            cx = (card.getValue()-1)*79;
            switch (card.getSuit()) {
            case Card.CLUBS:
               cy = 0;
               break;
            case Card.DIAMONDS:
               cy = 123;
               break;
            case Card.HEARTS:
               cy = 2*123;
               break;
            default:  // spades
               cy = 3*123;
               break;
            }
         }
         g.drawImage(cardImages,x,y,x+79,y+123,cx,cy,cx+79,cy+123,this);
      }
```

I will tell you later in this section how the image file, `cards.png`, can be loaded into the program.

* * *

In addition to images loaded from files, it is possible to create images by drawing to an off-screen canvas. An off-screen canvas can be represented by an object belonging to the class *BufferedImage*, which is defined in the package `java.awt.image`. *BufferedImage* is a subclass of *Image*, so that once you have a *BufferedImage*, you can copy it into a graphics context `g` using one of the `g.drawImage()` methods, just as you would do with any other image. A *BufferedImage* can be created using the constructor

```
public BufferedImage(int width, int height, int imageType)
```

where `width` and `height` specify the width and height of the image in pixels, and `imageType` can be one of several constants that are defined in the *BufferedImage*. The image type specifies how the color of each pixel is represented. The most likely value for `imageType` is `BufferedImage.TYPE_INT_RGB`, which specifies that the color of each pixel is a usual RGB color, with red, green and blue components in the range 0 to 255. The image type `BufferedImage.TYPE_INT_ARGB` represents an RGB image with "transparency"; see the next section for more information on this. The image type `BufferedImage.TYPE_BYTE_GRAY` can be used to create a ***grayscale*** image in which the only possible colors are shades of gray.

To draw to a *BufferedImage*, you need a graphics context that is set up to do its drawing on the image. If `OSC` is of type *BufferedImage*, then the method

```
OSC.getGraphics()
```

returns an object of type *Graphics* that can be used for drawing on the image.

There are several reasons why a programmer might want to draw to an off-screen canvas. One is to simply keep a copy of an image that is shown on the screen. Remember that a picture that is drawn on a component can be lost, for example when the component is covered by another window. This means that you have to be able to redraw the picture on demand, and that in turn means keeping enough information around to enable you to redraw the picture. One way to do this is to keep a copy of the picture in an off-screen canvas. Whenever the on-screen picture needs to be redrawn, you just have to copy the contents of the off-screen canvas onto the screen. Essentially, the off-screen canvas allows you to save a copy of the color of every

individual pixel in the picture. The sample program *PaintWithOffScreenCanvas.java* is a little painting program that uses an off-screen canvas in this way. In this program, the user can draw curves, lines, and various shapes; a "Tool" menu allows the user to select the thing to be drawn. There is also an "Erase" tool and a "Smudge" tool that I will get to later. A *BufferedImage* is used to store the user's picture. When the user changes the picture, the changes are made to the image, and the changed image is then copied to the screen. No record is kept of the shapes that the user draws; the only record is the color of the individual pixels in the off-screen image. (You should contrast this with the program *SimplePaint2.java* in Subsection 7.3.4, where the user's drawing is recorded as a list of objects that represent the shapes that the user drew.)

You should try the program (or the applet version in the on-line version of this section). Try drawing a Filled Rectangle on top of some other shapes. As you drag the mouse, the rectangle stretches from the starting point of the mouse drag to the current mouse location. As the mouse moves, the underlying picture seems to be unaffected—parts of the picture can be covered up by the rectangle and later uncovered as the mouse moves, and *they are still there*. What this means is that the rectangle that is shown as you drag the mouse can't actually be part of the off-screen canvas, since drawing something into an image means changing the color of some pixels in the image. The previous colors of those pixels are not stored anywhere else and so are permanently lost. In fact, when you draw a line, rectangle, or oval in `PaintWithOffScreenCanvas`, the shape that is shown as you drag the mouse is not drawn to the off-screen canvas at all. Instead, the `paintComponent()` method draws the shape on top of the contents of the canvas. Only when you release the mouse does the shape become a permanent part of the off-screen canvas. This illustrates the point that when an off-screen canvas is used, not everything that is visible on the screen has to be drawn on the canvas. Some extra stuff can be drawn on top of the contents of the canvas by the `paintComponent()` method. The other tools are handled differently from the shape tools. For the curve, erase, and smudge tools, the changes are made to the canvas immediately, as the mouse is being dragged.

Let's look at how an off-screen canvas is used in this program. The canvas is represented by an instance variable, `OSC`, of type *BufferedImage*. The size of the canvas must be the same size as the panel on which the canvas is displayed. The size can be determined by calling the `getWidth()` and `getHeight()` instance methods of the panel. Furthermore, when the canvas is first created, it should be filled with the background color, which is represented in the program by an instance variable named `fillColor`. All this is done by the method:

```
/**
 * This method creates the off-screen canvas and fills it with the current
 * fill color.
 */
private void createOSC() {
   OSC = new BufferedImage(getWidth(),getHeight(),BufferedImage.TYPE_INT_RGB);
   Graphics osg = OSC.getGraphics();
   osg.setColor(fillColor);
   osg.fillRect(0,0,getWidth(),getHeight());
   osg.dispose();
}
```

Note how it uses `OSC.getGraphics()` to obtain a graphics context for drawing to the image. Also note that the graphics context is disposed at the end of the method. It is good practice to dispose a graphics context when you are finished with it. There still remains the problem of where to call this method. The problem is that the width and height of the panel object are not set until some time after the panel object is constructed. If `createOSC()` is called in

the constructor, `getWidth()` and `getHeight()` will return the value zero and we won't get an off-screen image of the correct size. The approach that I take in `PaintWithOffScreenCanvas` is to call `createOSC()` in the `paintComponent()` method, the first time the `paintComponent()` method is called. At that time, the size of the panel has definitely been set, but the user has not yet had a chance to draw anything. With this in mind you are ready to understand the `paintComponent()` method:

```
public void paintComponent(Graphics g) {

   /* First create the off-screen canvas, if it does not already exist. */

   if (OSC == null)
      createOSC();

   /* Copy the off-screen canvas to the panel.  Since we know that the
      image is already completely available, the fourth "ImageObserver"
      parameter to g.drawImage() can be null.  Since the canvas completely
      fills the panel, there is no need to call super.paintComponent(g). */

   g.drawImage(OSC,0,0,null);

   /* If the user is currently dragging the mouse to draw a line, oval,
      or rectangle, draw the shape on top of the image from the off-screen
      canvas, using the current drawing color.  (This is not done if the
      user is drawing a curve or using the smudge tool or the erase tool.) */

   if (dragging && SHAPE_TOOLS.contains(currentTool)) {
      g.setColor(currentColor);
      putCurrentShape(g);
   }

}
```

Here, `dragging` is a **boolean** instance variable that is set to true while the user is dragging the mouse, and `currentTool` tells which tool is currently in use. The possible tools are defined by an `enum` named *Tool*, and `SHAPE_TOOLS` is a variable of type *EnumSet<Tool>* that contains the line, oval, rectangle, filled oval, and filled rectangle tools. (See Subsection 10.2.4.)

You might notice that there is a problem if the size of the panel is ever changed, since the size of the off-screen canvas will not be changed to match. The `PaintWithOffScreenCanvas` program does not allow the user to resize the program's window, so this is not an issue in that program. If we want to allow resizing, however, a new off-screen canvas must be created whenever the size of the panel changes. One simple way to do this is to check the size of the canvas in the `paintComponent()` method and to create a new canvas if the size of the canvas does not match the size of the panel:

```
if (OSC == null || getWidth() != OSC.getWidth() || getHeight() != OSC.getHeight())
   createOSC();
```

Of course, this will discard the picture that was contained in the old canvas unless some arrangement is made to copy the picture from the old canvas to the new one before the old canvas is discarded.

The other point in the program where the off-screen canvas is used is during a mouse-drag operation, which is handled in the `mousePressed()`, `mouseDragged()`, and `mouseReleased()` methods. The strategy that is implemented was discussed above. Shapes are drawn to the off-screen canvas only at the end of the drag operation, in the `mouseReleased()` method. However,

as the user drags the mouse, the part of the image over which the shape appears is re-copied from the canvas onto the screen each time the mouse is moved. Then the `paintComponent()` method draws the shape that the user is creating on top of the image from the canvas. For the non-shape (curve and smudge) tools, on the other hand, changes are made directly to the canvas, and the region that was changed is repainted so that the change will appear on the screen. (By the way, the program uses a version of the `repaint()` method that repaints just a part of a component. The command `repaint(x,y,width,height)` tells the system to repaint the rectangle with upper left corner `(x,y)` and with the specified width and height. This can be substantially faster than repainting the entire component.) See the source code, *PaintWithOffScreenCanvas.java*, if you want to see how it's all done.

* * *

One traditional use of off-screen canvasses is for ***double buffering***. In double-buffering, the off-screen image is an exact copy of the image that appears on screen; whenever the on-screen picture needs to be redrawn, the new picture is drawn step-by-step to an off-screen image. This can take some time. If all this drawing were done on screen, the user might see the image flicker as it is drawn. Instead, the long drawing process takes place off-screen and the completed image is then copied very quickly onto the screen. The user doesn't see all the steps involved in redrawing. This technique can be used to implement smooth, flicker-free animation.

The term "double buffering" comes from the term "frame buffer," which refers to the region in memory that holds the image on the screen. In fact, true double buffering uses two frame buffers. The video card can display either frame buffer on the screen and can switch instantaneously from one frame buffer to the other. One frame buffer is used to draw a new image for the screen. Then the video card is told to switch from one frame buffer to the other. No copying of memory is involved. Double-buffering as it is implemented in Java does require copying, which takes some time and is not perfectly flicker-free.

In Java's older AWT graphical API, it was up to the programmer to do double buffering by hand. In the Swing graphical API, double buffering is applied automatically by the system, and the programmer doesn't have to worry about it. (It is possible to turn this automatic double buffering off in Swing, but there is seldom a good reason to do so.)

One final historical note about off-screen canvasses: There is an alternative way to create them. The *Component* class defines the following instance method, which can be used in any GUI component object:

```
public Image createImage(int width, int height)
```

This method creates an *Image* with a specified width and height. You can use this image as an off-screen canvas in the same way that you would a *BufferedImage*. In fact, you can expect that in a modern version of Java, the image that is returned by this method is in fact a *BufferedImage*. The `createImage()` method was part of Java from the beginning, before the *BufferedImage* class was introduced.

13.1.2 Working With Pixels

One good reason to use a *BufferedImage* is that it allows easy access to the colors of individual pixels. If `image` is of type *BufferedImage*, then we have the methods:

- `image.getRGB(x,y)` — returns an **int** that encodes the color of the pixel at coordinates `(x,y)` in the image. The values of the integers `x` and `y` must lie within the image. That is,

it must be true that `0 <= x < image.getWidth()` and `0 <= y < image.getHeight()`; if not, then an exception is thrown.

- `image.setRGB(x,y,rgb)` — sets the color of the pixel at coordinates `(x,y)` to the color encoded by `rgb`. Again, `x` and `y` must be in the valid range. The third parameter, `rgb`, is an integer that encodes the color.

These methods use integer codes for colors. If `c` is of type *Color*, the integer code for the color can be obtained by calling `c.getRGB()`. Conversely, if `rgb` is an integer that encodes a color, the corresponding *Color* object can be obtained with the constructor call `new Color(rgb)`. This means that you can use

```
Color c = new Color( image.getRGB(x,y) )
```

to get the color of a pixel as a value of type *Color*. And if `c` is of type *Color*, you can set a pixel to that color with

```
image.setRGB( x, y, c.getRGB() );
```

The red, green, and blue components of a color are represented as 8-bit integers, in the range 0 to 255. When a color is encoded as a single **int**, the blue component is contained in the eight low-order bits of the **int**, the green component in the next lowest eight bits, and the red component in the next eight bits. (The eight high order bits store the "alpha component" of the color, which we'll encounter in the next section.) It is easy to translate between the two representations using the ***shift operators*** `<<` and `>>` and the ***bitwise logical operators*** `&` and `|`. (I have not covered these operators previously in this book. Briefly: If `A` and `B` are integers, then `A << B` is the integer obtained by shifting each bit of `A`, `B` bit positions to the left; `A >> B` is the integer obtained by shifting each bit of `A`, `B` bit positions to the right; `A & B` is the integer obtained by applying the logical **and** operation to each pair of bits in `A` and `B`; and `A | B` is obtained similarly, using the logical **or** operation. For example, using 8-bit binary numbers, we have: `01100101 & 10100001` is `00100001`, while `01100101 | 10100001` is `11100101`.) You don't necessarily need to understand these operators. Here are incantations that you can use to work with color codes:

```
/* Suppose that rgb is an int that encodes a color.
   To get separate red, green, and blue color components: */

int red = (rgb >> 16) & 0xFF;
int green = (rgb >> 8) & 0xFF;
int blue = rgb & 0xFF;

/* Suppose that red, green, and blue are color components in
   the range 0 to 255.  To combine them into a single int: */

int rgb = (red << 16) | (green << 8) | blue;
```

* * *

An example of using pixel colors in a *BufferedImage* is provided by the smudge tool in the sample program *PaintWithOffScreenCanvas.java*. The purpose of this tool is to smear the colors of an image, as if it were drawn in wet paint. For example, if you rub the middle of a black rectangle with the smudge tool, you'll get something like this:

This is an effect that can only be achieved by manipulating the colors of individual pixels! Here's how it works: when the user presses the mouse using the smudge tool, the color components of a 7-by-7 block of pixels are copied from the off-screen canvas into arrays named `smudgeRed`, `smudgeGreen` and `smudgeBlue`. This is done in the `mousePressed()` routine with the following code:

```
int w = OSC.getWidth();
int h = OSC.getHeight();
int x = evt.getX();
int y = evt.getY();
for (int i = 0; i < 7; i++)
   for (int j = 0; j < 7; j++) {
      int r = y + j - 3;
      int c = x + i - 3;
      if (r < 0 || r >= h || c < 0 || c >= w) {
            // A -1 in the smudgeRed array indicates that the
            // corresponding pixel was outside the canvas.
         smudgeRed[i][j] = -1;
      }
      else {
         int color = OSC.getRGB(c,r);
         smudgeRed[i][j] = (color >> 16) & 0xFF;
         smudgeGreen[i][j] = (color >> 8) & 0xFF;
         smudgeBlue[i][j] = color & 0xFF;
      }
   }
```

The arrays are of type `double[][]` because I am going to do some computations with them that require real numbers. As the user moves the mouse, the colors in the array are blended with the colors in the image, just as if you were mixing wet paint by smudging it with your finger. That is, the colors at the new mouse position in the image are replaced with a weighted average of the current colors in the image and the colors in the arrays. This has the effect of moving some of the color from the previous mouse position to the new mouse position. At the same time, the colors in the arrays are replaced by a weighted average of the old colors in the arrays and the colors from the image. This has the effect of moving some color from the image into the arrays. This is done using the following code for each pixel position, `(c,r)`, in a 7-by-7 block around the new mouse location:

```
int curCol = OSC.getRGB(c,r);
int curRed = (curCol >> 16) & 0xFF;
int curGreen = (curCol >> 8) & 0xFF;
int curBlue = curCol & 0xFF;
int newRed = (int)(curRed*0.7 + smudgeRed[i][j]*0.3);
int newGreen = (int)(curGreen*0.7 + smudgeGreen[i][j]*0.3);
int newBlue = (int)(curBlue*0.7 + smudgeBlue[i][j]*0.3);
int newCol = newRed << 16 | newGreen << 8 | newBlue;
OSC.setRGB(c,r,newCol);
```

```
smudgeRed[i][j] = curRed*0.3 + smudgeRed[i][j]*0.7;
smudgeGreen[i][j] = curGreen*0.3 + smudgeGreen[i][j]*0.7;
smudgeBlue[i][j] = curBlue*0.3 + smudgeBlue[i][j]*0.7;
```

13.1.3 Resources

Throughout this textbook, up until now, we have been thinking of a program as made up entirely of Java code. However, programs often use other types of data, including images, sounds, and text, as part of their basic structure. These data are referred to as ***resources***. An example is the image file, `cards.png`, that was used in the *HighLowWithImages.java* program earlier in this section. This file is part of the program. The program needs it in order to run. The user of the program doesn't need to know that this file exists or where it is located; as far as the user is concerned, it is just part of the program. The program of course, does need some way of locating the resource file and loading its data.

Resources are ordinarily stored in files that are in the same locations as the compiled class files for the program. Class files are located and loaded by something called a ***class loader***, which is represented in Java by an object of type *ClassLoader*. A class loader has a list of locations where it will look for class files. This list is called the ***class path***. It includes the location where Java's standard classes are stored. It generally includes the current directory. If the program is stored in a jar file, the jar file is included on the class path. In addition to class files, a *ClassLoader* is capable of finding resource files that are located on the class path or in subdirectories of locations that are on the class path.

The first step in using a resource is to obtain a *ClassLoader* and to use it to locate the resource file. In the `HighLowWithImages` program, this is done with:

```
ClassLoader cl = HighLowWithImages.class.getClassLoader();
URL imageURL = cl.getResource("cards.png");
```

The idea of the first line is that in order to get a class loader, you have to ask a class that was loaded by the class loader. Here, `HighLowWithImages.class` is a name for the object that represents the actual class, *HighLowWithImages*. In other programs, you would just substitute for "`HighLowWithImages`" the name of the class that contains the call to `getClassLoader()`. Alternatively, if `obj` is any object, then you can obtain a class loader by calling `obj.getClass().getClassLoader()`.

The second line in the above code uses the class loader to locate the resource file named `cards.png`. The return value of `cl.getResource()` is of type `java.net.URL`, and it represents the location of the resource rather than the resource itself. If the resource file cannot be found, then the return value is null. The class *URL* was discussed in Subsection 11.4.1.

Often, resources are stored not directly on the class path but in a subdirectory. In that case, the parameter to `getResource()` must be a path name that includes the directory path to the resource. For example, suppose that the image file "cards.png" were stored in a directory named `images` inside a directory named `resources`, where `resources` is directly on the class path. Then the path to the file is "resources/images/cards.png" and the command for locating the resource would be

```
URL imageURL = cl.getResource("resources/images/cards.png");
```

Once you have a *URL* that represents the location of a resource file, you could use a *URLConnection*, as discussed in Subsection 11.4.1, to read the contents of that file. However, Java provides more convenient methods for loading several types of resources. For loading image

resources, a convenient method is available in the class `java.awt.Toolkit`. It can be used as in the following line from `HighLowWithImages`, where `cardImages` is an instance variable of type *Image* and `imageURL` is the *URL* that represents the location of the image file:

```
cardImages = Toolkit.getDefaultToolkit().createImage(imageURL);
```

This still does not load the image completely—that will only be done later, for example when `cardImages` is used in a `drawImage` command. Another technique, which does read the image completely, is to use the `ImageIO.read()` method, which will be discussed in Subsection 13.1.5

* * *

The *Applet* and *JApplet* classes have an instance method that can be used to load an image from a given *URL*:

```
public Image getImage(URL imageURL)
```

When you are writing an applet, this method can be used as yet another technique for loading an image resource.

More interesting is the fact that *Applet* and *JApplet* contain a `static` method that can be used to load sound resources:

```
public static AudioClip newAudioClip(URL soundURL)
```

Since this is a `static` method, it can be used in any program, not just in applets, simply by calling it as `Applet.newAudioClip(soundURL)` or `JApplet.newAudioClip(soundURL)`. (This seems to be the only easy way to use sounds in a Java program; it's not clear why this capability is only in the applet classes.) The return value is of type `java.applet.AudioClip`. Once you have an *AudioClip*, you can call its `play()` method to play the audio clip from the beginning.

Here is a method that puts all this together to load and play the sound from an audio resource file:

```
private void playAudioResource(String audioResourceName) {
   ClassLoader cl = SoundAndCursorDemo.class.getClassLoader();
   URL resourceURL = cl.getResource(audioResourceName);
   if (resourceURL != null) {
      AudioClip sound = JApplet.newAudioClip(resourceURL);
      sound.play();
   }
}
```

This method is from a sample program `SoundAndCursorDemo` that will be discussed in the next subsection. Of course, if you plan to reuse the sound often, it would be better to load the sound once into an instance variable of type *AudioClip*, which could then be used to play the sound any number of times, without the need to reload it each time.

The *AudioClip* class supports audio files in the common WAV, AIFF, and AU formats.

13.1.4 Cursors and Icons

The position of the mouse is represented on the computer's screen by a small image called a ***cursor***. In Java, the cursor is represented by an object of type `java.awt.Cursor`. A *Cursor* has an associated image. It also has a ***hot spot***, which is a *Point* that specifies the pixel within the image that corresponds to the exact position on the screen where the mouse is pointing. For example, for a typical "arrow" cursor, the hot spot is the tip of the arrow. For a "crosshair" cursor, the hot spot is the center of the crosshairs.

The *Cursor* class defines several standard cursors, which are identified by constants such as `Cursor.CROSSHAIR_CURSOR` and `Cursor.DEFAULT_CURSOR`. You can get a standard cursor by calling the `static` method `Cursor.getPredefinedCursor(code)`, where `code` is one of the constants that identify the standard cursors. It is also possible to create a custom cursor from an *Image*. The *Image* might be obtained as an image resource, as described in the previous subsection. It could even be a *BufferedImage* that you create in your program. It should be small, maybe 16-by-16 or 24-by-24 pixels. (Some platforms might only be able to handle certain cursor sizes; see the documentation for `Toolkit.getBestCursorSize()` for more information.) A custom cursor can be created by calling the `static` method `createCustomCursor()` in the *Toolkit* class:

```
Cursor c = Toolkit.getDefaultToolkit().createCustomCursor(image,hotSpot,name);
```

where `hotSpot` is of type *Point* and `name` is a *String* that will act as a name for the cursor (and which serves no real purpose that I know of).

Cursors are associated with GUI components. When the mouse moves over a component, the cursor changes to whatever *Cursor* is associated with that component. To associate a *Cursor* with a component, call the component's instance method `setCursor(cursor)`. For example, to set the cursor for a *JPanel*, `panel`, to be the standard "wait" cursor:

```
panel.setCursor( Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR) );
```

To reset the cursor to be the default cursor, you can use:

```
panel.setCursor( Cursor.getDefaultCursor() );
```

To set the cursor to be an image that is defined in an image resource file named `imageResource`, you might use:

```
ClassLoader cl = SoundAndCursorDemo.class.getClassLoader();
URL resourceURL = cl.getResource(imageResource);
if (resourceURL != null) {
   Toolkit toolkit = Toolkit.getDefaultToolkit();
   Image image = toolkit.createImage(resourceURL);
   Point hotSpot = new Point(7,7);
   Cursor cursor = toolkit.createCustomCursor(image, hotSpot, "mycursor");
   panel.setCursor(cursor);
}
```

The sample program *SoundAndCursorDemo.java* shows how to use predefined and custom cursors and how to play sounds from resource files. The program has several buttons that you can click. Some of the buttons change the cursor that is associated with the main panel of the program. Some of the buttons play sounds. When you play a sound, the cursor is reset to be the default cursor. You can find an applet version of the program in the on-line version of this section.

Another standard use of images in GUI interfaces is for ***icons***. An icon is simply a small picture. As we'll see in Section 13.3, icons can be used on Java's buttons, menu items, and labels; in fact, for our purposes, an icon is simply an image that can be used in this way.

An icon is represented by an object of type *Icon*, which is actually an `interface` rather than a class. The class *ImageIcon*, which implements the *Icon* interface, is used to create icons from *Images*. If `image` is a (rather small) *Image*, then the constructor call `new ImageIcon(image)` creates an *ImageIcon* whose picture is the specified image. Often, the image comes from a resource file. We will see examples of this later in this chapter

13.1.5 Image File I/O

The class `javax.imageio.ImageIO` makes it easy to save images from a program into files and to read images from files into a program. This would be useful in a program such as `PaintWithOffScreenCanvas`, so that the users would be able to save their work and to open and edit existing images. (See Exercise 13.1.)

There are many ways that the data for an image could be stored in a file. Many standard formats have been created for doing this. Java supports at least three standard image formats: PNG, JPEG, and GIF. (Individual implementations of Java might support more.) The JPEG format is "lossy," which means that the picture that you get when you read a JPEG file is only an approximation of the picture that was saved. Some information in the picture has been lost. Allowing some information to be lost makes it possible to compress the image into a lot fewer bits than would otherwise be necessary. Usually, the approximation is quite good. It works best for photographic images and worst for simple line drawings. The PNG format, on the other hand is "lossless," meaning that the picture in the file is an exact duplicate of the picture that was saved. A PNG file is compressed, but not in a way that loses information. The compression works best for images made up mostly of large blocks of uniform color; it works **worst** for photographic images. GIF is an older format that is limited to just 256 colors in an image; it has mostly been superseded by PNG.

Suppose that `image` is a *BufferedImage*. The image can be saved to a file simply by calling

```
ImageIO.write( image, format, file )
```

where `format` is a *String* that specifies the image format of the file and `file` is a *File* that specifies the file that is to be written. (See Subsection 11.2.2 for information about the *File* class.) The `format` string should ordinarily be either `"PNG"` or `"JPEG"`, although other formats might be supported.

`ImageIO.write()` is a `static` method in the *ImageIO* class. It returns a **boolean** value that is `false` if the image format is not supported. That is, if the specified image format is not supported, then the image is **not** saved, but no exception is thrown. This means that you should always check the return value! For example:

```
boolean hasFormat = ImageIO.write(OSC,format,selectedFile);
if ( ! hasFormat )
   throw new Exception(format + " format is not available.");
```

If the image format **is** recognized, it is still possible that an *IOException* might be thrown when the attempt is made to send the data to the file.

Usually, the file to be used in `ImageIO.write()` will be selected by the user using a *JFileChooser*, as discussed in Subsection 11.2.3. For example, here is a typical method for saving an image. (The use of "this" as a parameter in several places assumes that this method is defined in a subclass of *JComponent*.)

```
/**
 * Attempts to save an image to a file selected by the user.
 * @param image the BufferedImage to be saved to the file
 * @param format the format of the image, probably either "PNG" or "JPEG"
 */
private void doSaveFile(BufferedImage image, String format) {
   if (fileDialog == null)
      fileDialog = new JFileChooser();
   fileDialog.setSelectedFile(new File("image." + format.toLowerCase()));
```

```
        fileDialog.setDialogTitle("Select File to be Saved");
        int option = fileDialog.showSaveDialog(this);
        if (option != JFileChooser.APPROVE_OPTION)
           return;  // User canceled or clicked the dialog's close box.
        File selectedFile = fileDialog.getSelectedFile();
        if (selectedFile.exists()) {  // Ask the user whether to replace the file.
           int response = JOptionPane.showConfirmDialog( null,
                 "The file \"" + selectedFile.getName()
                 + "\" already exists.\nDo you want to replace it?",
                 "Confirm Save",
                 JOptionPane.YES_NO_OPTION,
                 JOptionPane.WARNING_MESSAGE );
           if (response != JOptionPane.YES_OPTION)
              return;  // User does not want to replace the file.
        }
        try {
           boolean hasFormat = ImageIO.write(image,format,selectedFile);
           if ( ! hasFormat )
              throw new Exception(format + " format is not available.");
        }
        catch (Exception e) {
           JOptionPane.showMessageDialog(this,
                           "Sorry, an error occurred while trying to save image."));
           e.printStackTrace();
        }
     }
```

* * *

The *ImageIO* class also has a `static read()` method for reading an image from a file into a program. The method

```
ImageIO.read( inputFile )
```

takes a variable of type *File* as a parameter and returns a *BufferedImage*. The return value is `null` if the file does not contain an image that is stored in a supported format. Again, no exception is thrown in this case, so you should always be careful to check the return value. It is also possible for an *IOException* to occur when the attempt is made to read the file. There is another version of the `read()` method that takes an *InputStream* instead of a file as its parameter, and a third version that takes a *URL*.

Earlier in this section, we encountered another method for reading an image from a *URL*, the `createImage()` method from the *Toolkit* class. The difference is that `ImageIO.read()` reads the image data completely and stores the result in a *BufferedImage*. On the other hand, `createImage()` does not actually read the data; it really just stores the image location and the data won't be read until later, when the image is used. This has the advantage that the `createImage()` method itself can complete very quickly. `ImageIO.read()`, on the other hand, can take some time to execute.

13.2 Fancier Graphics

The graphics commands provided by the *Graphics* class are sufficient for many purposes. However, recent versions of Java provide a much larger and richer graphical toolbox in the form

of the class `java.awt.Graphics2D`. I mentioned *Graphics2D* in Subsection 6.3.5 and promised to discuss it further in this chapter.

Graphics2D is a subclass of *Graphics*, so all of the graphics commands that you already know can be used with a *Graphics2D* object. In fact, when you obtain a *Graphics* context for drawing on a Swing component or on a *BufferedImage*, the graphics object is actually of type *Graphics2D* and can be type-cast to gain access to the advanced *Graphics2D* graphics commands. Furthermore, *BufferedImage* has an instance method, `createGraphics()`, that returns a graphics context of type *Graphics2D*. For example, if `image` is of type *BufferedImage*, then you can get a *Graphics2D* for drawing on the image using:

```
Graphics2D g2 = image.createGraphics();
```

And, as mentioned in Subsection 6.3.5, to use *Graphics2D* commands in the `paintComponent()` method of a Swing component, you can write a `paintComponent()` method of the form:

```
public void paintComponent(Graphics g) {
   super.paintComponent(g);
   Graphics g2 = (Graphics2D)g;
      .
      .   // Draw to the component using g2 (and g).
      .
}
```

Note that when you do this, `g` and `g2` are just two variables that refer to the same object, so they both draw to the same drawing surface; `g2` just gives you access to methods that are defined in *Graphics2D* but not in *Graphics*. When properties of `g2`, such as drawing color, are changed, the changes also apply to `g`. By saying

```
Graphics2D g2 = (Graphics2D)g.create()
```

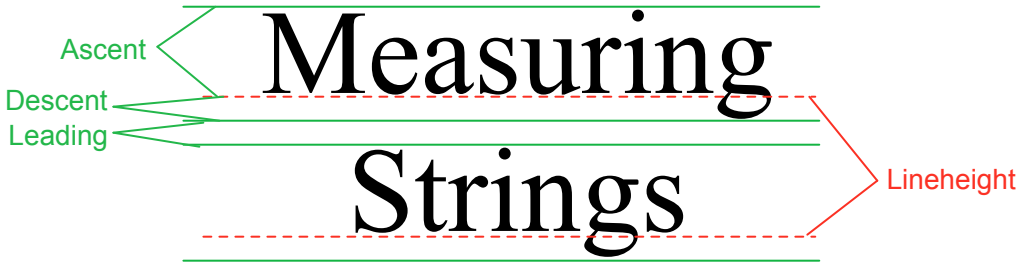
you can obtain a newly created graphics context. The object created by `g.create()` is a graphics context that draws to the same drawing surface as `g` and that initially has all the same properties as `g`. However, it is a separate object, so that changing properties in `g2` has no effect on `g`. This can be useful if you want to keep an unmodified copy of the original graphics context around for some drawing operations. (In this case, it is good practice to call `g2.dispose()` to dispose of the new graphics context when you are finished using it.)

13.2.1 Measuring Text

Although this section is mostly about *Graphics2D*, we start with a topic that has nothing to do with it.

Often, when drawing a string, it's important to know how big the image of the string will be. For example, you need this information if you want to center a string in a component. Or if you want to know how much space to leave between two lines of text, when you draw them one above the other. Or if the user is typing the string and you want to position a cursor at the end of the string. In Java, questions about the size of a string can be answered by an object belonging to the standard class `java.awt.FontMetrics`.

There are several lengths associated with any given font. Some of them are shown in this illustration:



The dashed lines in the illustration are the ***baselines*** of the two lines of text. The baseline of a string is the line on which the bases of the characters rest. The suggested distance between two baselines, for single-spaced text, is known as the ***lineheight*** of the font. The ***ascent*** is the distance that tall characters can rise above the baseline, and the ***descent*** is the distance that tails like the one on the letter "g" can descend below the baseline. The ascent and descent do not add up to the lineheight, because there should be some extra space between the tops of characters in one line and the tails of characters on the line above. The extra space is called ***leading***. (The term comes from the time when lead blocks were used for printing. Characters were formed on blocks of lead that were lined up to make up the text of a page, covered with ink, and pressed onto paper to print the page. Extra, blank "leading" was used to separate the lines of characters.) All these quantities can be determined by calling instance methods in a *FontMetrics* object. There are also methods for determining the width of a character and the total width of a string of characters.

Recall that a font in Java is represented by the class *Font*. A *FontMetrics* object is associated with a given font and is used to measure characters and strings in that font. If `font` is of type *Font* and `g` is a graphics context, you can get a *FontMetrics* object for the font by calling `g.getFontMetrics(font)`. If `fm` is the variable that refers to the *FontMetrics* object, then the ascent, descent, leading, and lineheight of the font can be obtained by calling `fm.getAscent()`, `fm.getDescent()`, `fm.getLeading()`, and `fm.getHeight()`. If `ch` is a character, then `fm.charWidth(ch)` is the width of the character when it is drawn in that font. If `str` is a string, then `fm.stringWidth(str)` is the width of the string when drawn in that font. For example, here is a `paintComponent()` method that shows the message "Hello World" in the exact center of the component:

```
public void paintComponent(Graphics g) {
   super.paintComponent(g);

   int strWidth, strHeight; // Width and height of the string.
   int centerX, centerY;    // Coordinates of the center of the component.
   int baseX, baseY;        // Coordinates of the basepoint of the string.
   int topOfString;         // y-coordinate of the top of the string.

   centerX = getWidth() / 2;
   centerY = getHeight() / 2;

   Font font = g.getFont();  // What font will g draw in?
   FontMetrics fm = g.getFontMetrics(font);
   strWidth = fm.stringWidth("Hello World");
   strHeight = fm.getAscent();  // Note: There are no tails on
                                //   any of the chars in the string!

   baseX = centerX - (strWidth/2);  // Move back from center by half the
```

```
                                          //    width of the string.

        topOfString = centerY - (strHeight/2);  // Move up from center by half
                                                //   the height of the string.

        baseY = topOfString + fm.getAscent();  // Baseline is fm.getAscent() pixels
                                               //   below the top of the string.

        g.drawString("Hello World", baseX, baseY); // Draw the string.
    }
```

You can change the font that is used for drawing strings as described in Subsection 6.3.3. For the height of the string in this method, I use `fm.getAscent()`. If I were drawing "Goodbye World" instead of "Hello World," I would have used `fm.getAscent() + fm.getDescent()`, where the descent is added to the height in order to take into account the tail on the "y" in "Goodbye". The value of `baseX` is computed to be the amount of space between the left edge of the component and the start of the string. It is obtained by subtracting half the width of the string from the horizontal center of the component. This will center the string horizontally in the component. The next line computes the position of the top of the string in the same way. However, to draw the string, we need the y-coordinate of the baseline, not the y-coordinate of the top of the string. The baseline of the string is below the top of the string by an amount equal to the ascent of the font.

There is an example of centering a two-line block of text in the sample program *TransparencyDemo.java*, which is discussed in the next subsection.

13.2.2 Transparency

A color is represented by red, blue, and green components. In Java's usual representation, each component is an eight-bit number in the range 0 to 255. The three color components can be packed into a 32-bit integer, but that only accounts for 24 bits in the integer. What about the other eight bits? They don't have to be wasted. They can be used as a fourth component of the color, the ***alpha component***. The alpha component can be used in several ways, but it is most commonly associated with ***transparency***. When you draw with a transparent color, it's like laying down a sheet of colored glass. It doesn't completely obscure the part of the image that is colored over. Instead, the background image is ***blended*** with the transparent color that is used for drawing—as if you were looking at the background through colored glass. This type of drawing is properly referred to as ***alpha blending***, and it is not equivalent to true transparency; nevertheless, most people refer to it as transparency.

The value of the alpha component determines how transparent that color is. Actually, the alpha component gives the ***opaqueness*** of the color. Opaqueness is the opposite of transparency. If something is fully opaque, you can't see through it at all; if something is almost fully opaque, then it is just a little transparent; and so on. When the alpha component of a color has the maximum possible value, the color is fully opaque. When you draw with a fully opaque color, that color simply replaces the color of the background over which you draw. This is the only type of color that we have used up until now. If the alpha component of a color is zero, then the color is perfectly transparent, and drawing with that color has no effect at all. Intermediate values of the alpha component give partially opaque colors that will blend with the background when they are used for drawing.

The sample program *TransparencyDemo.java* can help you to understand transparency. When you run the program you will see a display area containing a triangle, an oval, a rectangle, and some text. Sliders at the bottom of the applet allow you to control the degree of

transparency of each shape. When a slider is moved all the way to the right, the corresponding shape is fully opaque; all the way to the left, and the shape is fully transparent. An applet version of the program can be found in the on-line version of this section.

* * *

Colors with alpha components were introduced in Java along with *Graphics2D*, but they can be used with ordinary *Graphics* objects as well. To specify the alpha component of a color, you can create the *Color* object using one of the following constructors from the *Color* class:

```
public Color(int red, int green, int blue, int alpha);

public Color(float red, float green, float blue, float alpha);
```

In the first constructor, all the parameters must be integers in the range 0 to 255. In the second, the parameters must be in the range 0.0 to 1.0. For example,

```
Color transparentRed = new Color( 255, 0, 0, 200 );
```

makes a slightly transparent red, while

```
Color tranparentCyan = new Color( 0.0F, 1.0F, 1.0F, 0.5F);
```

makes a blue-green color that is 50% opaque. (The advantage of the constructor that takes parameters of type **float** is that it lets you think in terms of percentages.) When you create an ordinary RGB color, as in `new Color(255,0,0)`, you just get a fully opaque color.

Once you have a transparent color, you can use it in the same way as any other color. That is, if you want to use a *Color* `c` to draw in a graphics context `g`, you just say `g.setColor(c)`, and subsequent drawing operations will use that color. As you can see, transparent colors are very easy to use.

* * *

A *BufferedImage* with image type `BufferedImage.TYPE_INT_ARGB` can use transparency. The color of each pixel in the image can have its own alpha component, which tells how transparent that pixel will be when the image is drawn over some background. A pixel whose alpha component is zero is perfectly transparent, and has no effect at all when the image is drawn; in effect, it's not part of the image at all. It is also possible for pixels to be partly transparent. When an image is saved to a file, information about transparency might be lost, depending on the file format. The PNG image format supports transparency; JPEG does not. (If you look at the images of playing cards that are used in the program `HighLowWithImages` in Subsection 13.1.1, you might notice that the tips of the corners of the cards are fully transparent. The card images are from a PNG file, *cards.png*.)

An ARGB *BufferedImage* should be fully transparent when it is first created, but if you want to make sure, here is one way of doing so: The *Graphics2D* class has a method `setBackground()` that can be used to set a background color for the graphics context, and it has a `clearRect()` method that fills a rectangle with the current background color. To create a fully transparent image with width `w` and height `h`, you can use:

```
BufferedImage image = new BufferedImage(w, h, BufferedImage.TYPE_INT_ARGB);
Graphics2D g2 = (Graphics2D)image.getGraphics();
g2.setBackground(new Color(0,0,0,0));  // (The R, G, and B values don't matter.)
g2.clearRect(0, 0, w, h);
```

(Note that simply drawing with a transparent color will not make pixels in the image transparent. The alpha component of a *Color* makes the color transparent when it is used for drawing; it does not change the transparency of the pixels that are modified by the drawing operation.)

As an example, just for fun, here is a method that will set the cursor of a component to be a red square with a transparent interior:

```
private void useRedSquareCursor() {
   BufferedImage image = new BufferedImage(24,24,BufferedImage.TYPE_INT_ARGB);
   Graphics2D g2 = (Graphics2D)image.getGraphics();
   g2.setBackground(new Color(0,0,0,0));
   g2.clearRect(0, 0, 24, 24);  // (should not be necessary in a new image)
   g2.setColor(Color.RED);
   g2.drawRect(0,0,23,23);
   g2.drawRect(1,1,21,21);
   g2.dispose();
   Point hotSpot = new Point(12,12);
   Toolkit tk = Toolkit.getDefaultToolkit();
   Cursor cursor = tk.createCustomCursor(image,hotSpot,"square");
   setCursor(cursor);
}
```

13.2.3 Antialiasing

To draw a geometric figure such as a line or circle, you just have to color the pixels that are part of the figure, right? Actually, there is a problem with this. Pixels are little squares. Geometric figures, on the other hand, are made of geometric points that have no size at all. Think about drawing a circle, and think about a pixel on the boundary of that circle. The infinitely thin geometric boundary of the circle cuts through the pixel. Part of the pixel lies inside the circle, part lies outside. So, when we are filling the circle with color, do we color that pixel or not? A possible solution is to color the pixel if the geometric circle covers 50% or more of the pixel. Following this procedure, however, leads to a visual defect known as ***aliasing***. It is visible in images as a jaggedness or "staircasing" effect along the borders of curved shapes. Lines that are not horizontal or vertical also have a jagged, aliased appearance. (The term "aliasing" seems to refer to the fact that many different geometric points map to the same pixel. If you think of the real-number coordinates of a geometric point as a "name" for the pixel that contains that point, then each pixel has many different names or "aliases.")

It's not possible to build a circle out of squares, but there is a technique that can eliminate some of the jaggedness of aliased images. The technique is called ***antialiasing***. Antialiasing is based on transparency. The idea is simple: If 50% of a pixel is covered by the geometric figure that you are trying to draw, then color that pixel with a color that is 50% transparent. If 25% of the pixel is covered, use a color that is 75% transparent (25% opaque). If the entire pixel is covered by the figure, of course, use a color that is 100% opaque—antialiasing only affects pixels that are only partly covered by the geometric shape.

In antialiasing, the color that you are drawing with is blended with the original color of the pixel, and the amount of blending depends on the fraction of the pixel that is covered by the geometric shape. (The fraction is difficult to compute exactly, so in practice, various methods are used to approximate it.) Of course, you still don't get a picture of the exact geometric shape, but antialiased images do tend to look better than jagged, aliased images.

For an example, look at the image in the next subsection. Antialiasing is used to draw the panels in the second and third row of the image, but it is not used in the top row. You should note the jagged appearance of the lines and rectangles in the top row. (By the way, when antialiasing is applied to a line, the line is treated as a geometric rectangle whose width is equal to the size of one pixel.)

Antialiasing is supported in *Graphics2D*. By default, antialiasing is turned off. If g2 is a graphics context of type *Graphics2D*, you can turn on antialiasing in g2 by saying:

```
g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                                    RenderingHints.VALUE_ANTIALIAS_ON);
```

As you can see, this is only a "hint" that you would like to use antialiasing, and it is even possible that the hint will be ignored. However, it is likely that subsequent drawing operations in g2 will be antialiased. If you want to turn antialiasing off in g2, you should say:

```
g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                                    RenderingHints.VALUE_ANTIALIAS_OFF);
```

13.2.4 Strokes and Paints

When using the *Graphics* class, any line that you draw will be a solid line that is one pixel thick. The *Graphics2D* class makes it possible to draw a much greater variety of lines. You can draw lines of any thickness, and you can draw lines that are dotted or dashed instead of solid.

An object of type *Stroke* contains information about how lines should be drawn, including how thick the line should be and what pattern of dashes and dots, if any, should be used. Every *Graphics2D* has an associated *Stroke* object. The default *Stroke* draws a solid line of thickness one. To get lines with different properties, you just have to install a different stroke into the graphics context.

Stroke is an `interface`, not a class. The class *BasicStroke*, which implements the `Stroke` interface, is the one that is actually used to create stroke objects. For example, to create a stroke that draws solid lines with thickness equal to 3, use:

```
BasicStroke line3 = new BasicStroke(3);
```

If g2 is of type *Graphics2D*, the stroke can be installed in g2 by calling its `setStroke()` command:

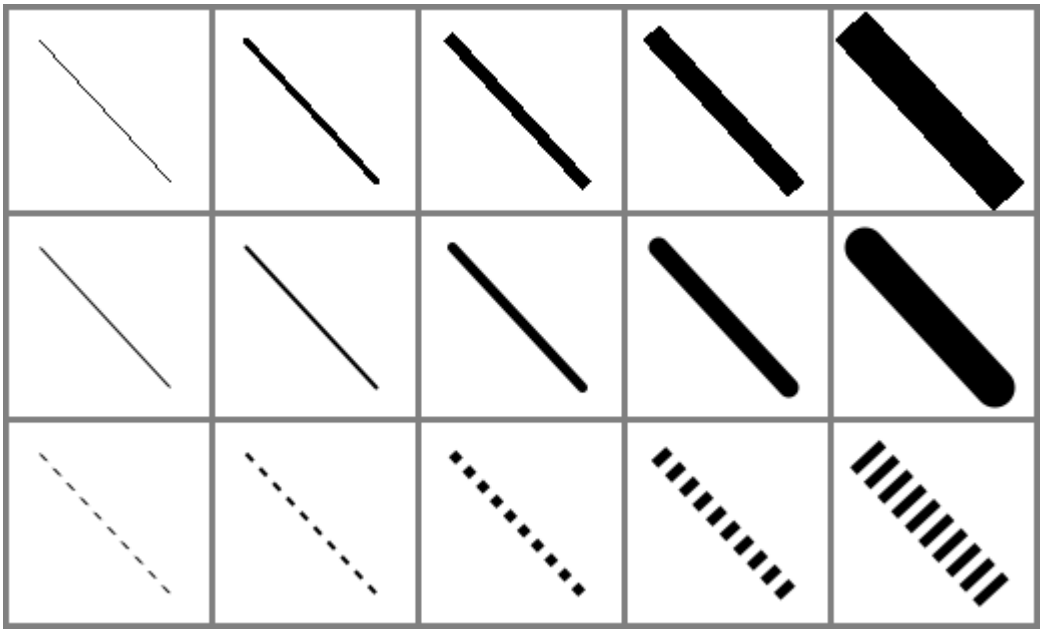
```
g2.setStroke(line3)
```

After calling this method, subsequent drawing operations will use lines that are three times as wide as the usual thickness. The thickness of a line can be given by a value of type **float**, not just by an **int**. For example, to use lines of thickness 2.5 in the graphics context g2, you can say:

```
g2.setStroke( new BasicStroke(2.5F) );
```

(Fractional widths make more sense if antialiasing is turned on.)

When you have a thick line, the question comes up, what to do at the ends of the line. If you draw a physical line with a large, round piece of chalk, the ends of the line will be rounded. When you draw a line on the computer screen, should the ends be rounded, or should the line simply be cut off flat? With the *BasicStroke* class, the choice is up to you. Maybe it's time to look at examples. This illustration shows fifteen lines, drawn using different *BasicStrokes*. Lines in the middle row have rounded ends; lines in the other two rows are simply cut off at their endpoints. Lines of various thicknesses are shown, and the bottom row shows dashed lines. (And, as mentioned above, only the bottom two rows are antialiased.)



This illustration shows the sample program *StrokeDemo.java*. (You can try an applet version of the program in the on-line version of this section.) In this program, you can click and drag in any of the small panels, and the lines in all the panels will be redrawn as you move the mouse. In addition, if you right-click and drag, then rectangles will be drawn instead of lines; this shows that strokes are used for drawing the outlines of shapes and not just for straight lines. If you look at the corners of the rectangles that are drawn by the program, you'll see that there are several ways of drawing a corner where two wide line segments meet.

All the options that you want for a *BasicStroke* have to be specified in the constructor. Once the stroke object is created, there is no way to change the options. There is one constructor that lets you specify all possible options:

```
public BasicStroke( float width, int capType, int joinType, float miterlimit,
                                                  float[] dashPattern, float dashPhase )
```

I don't want to cover all the options in detail, but here's some basic info:

- `width` specifies the thickness of the line
- `capType` specifies how the ends of a line are "capped." The possible values are `BasicStroke.CAP_SQUARE`, `BasicStroke.CAP_ROUND` and `BasicStroke.CAP_BUTT`. These values are used, respectively, in the first, second, and third rows of the above picture. The default is `BasicStroke.CAP_SQUARE`.
- `joinType` specifies how two line segments are joined together at corners. Possible values are `BasicStroke.JOIN_MITER`, `BasicStroke.JOIN_ROUND`, and `BasicStroke.JOIN_BEVEL`. Again, these are used in the three rows of panels in the sample program; the effect is only seen in the applet when drawing rectangles. The default is `BasicStroke.JOIN_MITER`.
- `miterLimit` is used only if the value of `joinType` is `JOIN_MITER`; just use the default value, `10.0F`.
- `dashPattern` is used to specify dotted and dashed lines. The values in the array specify lengths in the dot/dash pattern. The numbers in the array represent the length of a solid piece, followed by the length of a transparent piece, followed by the length of a solid piece,

and so on. At the end of the array, the pattern wraps back to the beginning of the array. If you want a solid line, use a different constructor that has fewer parameters.

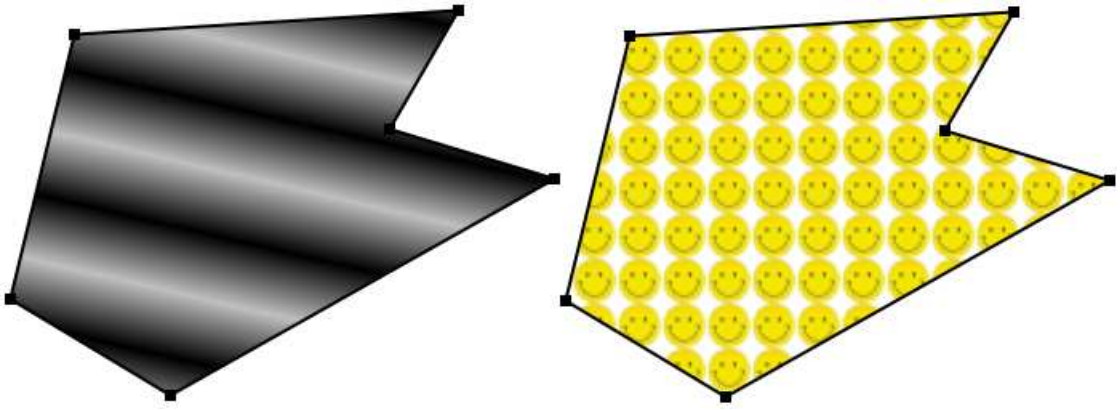
- `dashPhase` tells the computer where to start in the `dashPattern` array, for the first segment of the line. Use 0 for this parameter in most cases.

For the third row in the above picture, the `dashPattern` is set to `new float[] {5,5}`. This means that the lines are drawn starting with a solid segment of length 5, followed by a transparent section of length 5, and then repeating the same pattern. A simple dotted line would have thickness 1 and `dashPattern` `new float[] {1,1}`. A pattern of short and long dashes could be made by using `new float[] {10,4,4,4}`. For more information, see the Java documentation, or try experimenting with the source code for the sample program.

* * *

So now we can draw fancier lines. But any drawing operation is still restricted to drawing with a single color. We can get around that restriction by using *Paint*. An object of type *Paint* is used to assign color to each pixel that is "hit" by a drawing operation. *Paint* is an `interface`, and the *Color* class implements the *Paint* interface. When a color is used for painting, it applies the same color to every pixel that is hit. However, there are other types of paint where the color that is applied to a pixel depends on the coordinates of that pixel. Standard Java includes two classes that define paint with this property: *GradientPaint* and *TexturePaint*. In a ***gradient***, the color that is applied to pixels changes gradually from one color to a second color as you move from point to point. In a ***texture***, the pixel colors come from an image, which is repeated, if necessary, like a wallpaper pattern to cover the entire xy-plane.

It will be helpful to look at some examples. This illustration shows a polygon filled with two different paints. The polygon on the left uses a *GradientPaint* while the one on the right uses a *TexturePaint*. Note that in this picture, the paint is used only for filling the polygon. The outline of the polygon is drawn in a plain black color. However, *Paint* objects can be used for drawing lines as well as for filling shapes. These pictures were made by the sample program *PaintDemo.java*. In that program, you can select among several different paints, and you can control certain properties of the paints. As usual, an applet version of the program is available on line.



Basic gradient paints are created using the constructor

```
public GradientPaint(float x1, float y1, Color c1,
                     float x2, float y2, Color c2, boolean cyclic)
```

This constructs a gradient that has color `c1` at the point with coordinates `(x1,y1)` and color `c2` at the point `(x2,y2)`. As you move along the line between the two points, the color of the gradient changes from `c1` to `c2`; along lines perpendicular to this line, the color is constant. The last parameter, `cyclic`, tells what happens if you move past the point `(x2,y2)` on the line from `(x1,y1)` to `(x2,y2)`. If `cyclic` is `false`, the color stops changing and any point beyond `(x2,y2)` has color `c2`. If `cyclic` is `true`, then the colors continue to change in a cyclic pattern after you move past `(x2,y2)`. (It works the same way if you move past the other endpoint, `(x1,y1)`.) In most cases, you will set `cyclic` to `true`. Note that you can vary the points `(x1,y1)` and `(x2,y2)` to change the width and direction of the gradient. For example, to create a cyclic gradient that varies from black to light gray along the line from `(0,0)` to `(100,100)`, use:

```
new GradientPaint( 0, 0, Color.BLACK, 100, 100, Color.LIGHT_GRAY, true)
```

Java 6 introduced two new gradient paint classes, *LinearGradientPaint* and *RadialGradientPaint*. Linear gradient paints are similar to *GradientPaint* but can be based on more than two colors. Radial gradients color pixels based on their distance from a central point, which produces rings of constant color instead of lines of constant color. See the API documentation for details.

To construct a *TexturePaint*, you need a *BufferedImage* that contains the image that will be used for the texture. You also specify a rectangle in which the image will be drawn. The image will be scaled, if necessary, to exactly fill the rectangle. Outside the specified rectangle, the image will be repeated horizontally and vertically to fill the plane. You can vary the size and position of the rectangle to change the scale of the texture and its positioning on the plane. Ordinarily, however the upper left corner of the rectangle is placed at `(0,0)`, and the size of the rectangle is the same as the actual size of the image. The constructor for *TexturePaint* is defined as

```
public TexturePaint( BufferedImage textureImage, Rectangle2D anchorRect)
```

The *Rectangle2D* is part of the *Graphics2D* framework and will be discussed at the end of this section. Often, a call to the constructor takes the form:

```
new TexturePaint( image,
            new Rectangle2D.Double(0,0,image.getWidth(),image.getHeight() )
```

Once you have a *Paint* object, you can use the `setPaint()` method of a *Graphics2D* object to install the paint in a graphics context. For example, if `g2` is of type *Graphics2D*, then the command

```
g2.setPaint( new GradientPaint(0,0,Color.BLUE,100,100,Color.GREEN,true) );
```

sets up `g2` to use a gradient paint. Subsequent drawing operations with `g2` will draw using a blue/green gradient.

13.2.5 Transforms

In the standard drawing coordinates on a component, the upper left corner of the component has coordinates `(0,0)`. Coordinates are integers, and the coordinates `(x,y)` refer to the point that is `x` pixels over from the left edge of the component and `y` pixels down from the top. With *Graphics2D*, however, you are not restricted to using these coordinates. In fact, you can can set up a *Graphics2D* graphics context to use any system of coordinates that you like. You can use this capability to select the coordinate system that is most appropriate for the things that

you want to draw. For example, if you are drawing architectural blueprints, you might use coordinates in which one unit represents an actual distance of one foot.

Changes to a coordinate system are referred to as ***transforms***. There are three basic types of transform. A ***translate*** transform changes the position of the origin, `(0,0)`. A ***scale*** transform changes the scale, that is, the unit of distance. And a ***rotation*** transform applies a rotation about some point. You can make more complex transforms by combining transforms of the three basic types. For example, you can apply a rotation, followed by a scale, followed by a translation, followed by another rotation. When you apply several transforms in a row, their effects are cumulative. It takes a fair amount of study to fully understand complex transforms. I will limit myself here to discussing a few of the most simple cases, just to give you an idea of what transforms can do.

Suppose that `g2` is of type *Graphics2D*. Then `g2.translate(x,y)` moves the origin, `(0,0)`, to the point `(x,y)`. This means that if you use coordinates `(0,0)` *after* saying `g2.translate(x,y)`, then you are referring to the point that *used to be* `(x,y)`, before the translation was applied. All other coordinate pairs are moved by the same amount. For example saying

```
g.translate(x,y);
g.drawLine( 0, 0, 100, 200 );
```

draws the same line as

```
g.drawLine( x, y, 100+x, 200+y );
```

In the second case, you are just doing the same translation "by hand." A translation (like all transforms) affects all subsequent drawing operations. Instead of thinking in terms of coordinate systems, you might find it clearer to think of what happens to the objects that are drawn. After you say `g2.translate(x,y)`, any objects that you draw are displaced `x` units horizontally and `y` units vertically. Note that the parameters `x` and `y` can be real numbers.

As an example, perhaps you would prefer to have `(0,0)` at the center of a component, instead of at its upper left corner. To do this, just use the following command in the `paintComponent()` method of the component:

```
g2.translate( getWidth()/2, getHeight()/2 );
```

To apply a scale transform to a *Graphics2D* `g2`, use `g2.scale(s,s)`, where `s` is the real number that specifies the scaling factor. If `s` is greater than 1, everything is magnified by a factor of `s`, while if `s` is between 0 and 1, everything is shrunk by a factor of `s`. The center of scaling is `(0,0)`. That is, the point `(0,0)` is unaffected by the scaling, and other points more towards or away from `(0,0)` by a factor of `s`. Again, it can be clearer to think of the effect on objects that are drawn after a scale transform is applied. Those objects will be magnified or shrunk by a factor of `s`. Note that scaling affects **everything**, including thickness of lines and size of fonts. It is possible to use different scale factors in the horizontal and vertical direction with a command of the form `g2.scale(sx,sy)`, although that will distort the shapes of objects. By the way, it is even possible to use scale factors that are less than 0, which results in reflections. For example, after calling `g2.scale(-1,1)`, objects will be reflected horizontally through the line `x=0`.

The third type of basic transform is rotation. The command `g2.rotate(r)` rotates all subsequently drawn objects through an angle of `r` about the point `(0,0)`. You can rotate instead about the point `(x,y)` with the command `g2.rotate(r,x,y)`. All the parameters can be real numbers. Angles are measured in radians, where π radians are equal to 180 degrees. To rotate through an angle of `d` degrees, use

```
g2.rotate( d * Math.PI / 180 );
```

Positive angles are clockwise rotations, while negative angles are counterclockwise (unless you have applied a negative scale factor, which reverses the orientation).

Rotation is not as common as translation or scaling, but there are a few things that you can do with it that can't be done any other way. For example, you can use it to draw an image "on the slant." Rotation also makes it possible to draw text that is rotated so that its baseline is slanted or even vertical. To draw the string "Hello World" with its basepoint at `(x,y)` and rising at an angle of 30 degrees, use:

```
g2.rotate( -30 * Math.PI / 180, x, y );
g2.drawString( "Hello World", x, y );
```

To draw the message vertically, with the **center** of its baseline at the point `(x,y)`, we can use *FontMetrics* to measure the string, and say:

```
FontMetrics fm = g2.getFontMetrics( g2.getFont() );
int baselineLength = fm.stringWidth("Hello World");
g2.rotate( -90 * Math.PI / 180, x, y);
g2.drawString( "Hello World", x - baselineLength/2, y );
```

* * *

The drawing operations in the *Graphics* class use integer coordinates only. *Graphics2D* makes it possible to use real numbers as coordinates. This becomes particularly important once you start using transforms, since after you apply a scale, a square of size one might cover many pixels instead of just a single pixel. Unfortunately, the designers of Java couldn't decide whether to use numbers of type **float** or **double** as coordinates, and their indecision makes things a little more complicated than they need to be. (My guess is that they really wanted to use **float**, since values of type float have enough accuracy for graphics and are probably used in the underlying graphical computations of the computer. However, in Java programming, it's easier to use **double** than **float**, so they wanted to make it possible to use **double** values too.)

To use real number coordinates, you have to use classes defined in the package `java.awt.geom`. Among the classes in this package are classes that represent geometric shapes such as lines and rectangles. For example, the class *Line2D* represents a line whose endpoints are given as real number coordinates. The unfortunate thing is that *Line2D* is an abstract class, which means that you can't create objects of type *Line2D* directly. However, *Line2D* has two concrete subclasses that can be used to create objects. One subclass uses coordinates of type **float**, and one uses coordinates of type **double**. The most peculiar part is that these subclasses are defined as static nested classes inside *Line2D*. Their names are *Line2D.Float* and *Line2D.Double*. This means that *Line2D* objects can be created, for example, with:

```
Line2D line1 = new Line2D.Float( 0.17F, 1.3F, -2.7F, 5.21F );
Line2D line2 = new Line2D.Double( 0, 0, 1, 0);
Line2D line3 = new Line2D.Double( x1, y1, x2, y2 );
```

where `x1`, `y1`, `x2`, `y2` are any numeric variables. In my own code, I generally use *Line2D.Double* rather than *Line2D.Float*.

Other shape classes in `java.awt.geom` are similar. The class that represents rectangles is *Rectangle2D*. To create a rectangle object, you have to use either *Rectangle2D.Float* or *Rectangle2D.Double*. For example,

```
Rectangle2D rect = new Rectangle2D.Double( -0.5, -0.5, 1.0, 1.0 );
```

creates a rectangle with a corner at `(-0.5,-0.5)` and with width and height both equal to 1. Other classes include *Point2D*, which represents a single point; *Ellipse2D*, which represents an oval; and *Arc2D*, which represents an arc of a circle.

If `g2` is of type *Graphics2D* and `shape` is an object belonging to one of the 2D shape classes, then the command

```
g2.draw(shape);
```

draws the shape. For a shape such as a rectangle or ellipse that has an interior, only the outline is drawn. To fill in the interior of such a shape, use

```
g2.fill(shape)
```

For example, to draw a line from `(x1,y1)` to `(x2,y2)`, use

```
g2.draw( new Line2D.Double(x1,y1,x2,y2) );
```

and to draw a filled rectangle with a corner at `(3.5,7)`, with width 5 and height 3, use

```
g2.fill( new Rectangle2D.Double(3.5, 7, 5, 3) );
```

The package `java.awt.geom` also has a very nice class *GeneralPath* that can be used to draw polygons and curves defined by any number of points. See the Java documentation if you want to find out how to use it. In Java 6, *GeneralPath* has been largely superseded by *Path2D* which provides the same functionality but more closely follows the conventions used by other shape classes.

This section has introduced you to many of the interesting features of *Graphics2D*, but there is still a large part of the *Graphics2D* framework for you to explore.

13.3 Actions and Buttons

For the past two sections, we have been looking at some of the more advanced aspects of the Java graphics API. But the heart of most graphical user interface programming is using GUI components. In this section and the next, we'll be looking at *JComponents*. We'll cover several component classes that were not covered in Chapter 6, as well as some additional features of classes that were covered there.

This section is mostly about buttons. Buttons are among the simplest of GUI components, and it seems like there shouldn't be all that much to say about them. However, buttons are not as simple as they seem. For one thing, there are many different types of buttons. The basic functionality of buttons in Java is defined by the class `javax.swing.AbstractButton`. Subclasses of this class represent push buttons, check boxes, and radio buttons. Menu items are also considered to be buttons. The *AbstractButton* class defines a surprisingly large API for controlling the appearance of buttons. This section will cover part of that API, but you should see the class documentation for full details.

In this section, we'll also encounter a few classes that do not themselves define buttons but that are related to the button API, starting with "actions."

13.3.1 Action and AbstractAction

The *JButton* class represents push buttons. Up until now, we have created push buttons using the constructor

```
public JButton(String text);
```

which specifies text that will appear on the button. We then added an *ActionListener* to the button, to respond when the user presses it. Another way to create a *JButton* is using an *Action*. The *Action* interface represents the general idea of some action that can be performed, together with properties associated with that action, such as a name for the action, an icon that represents the action, and whether the action is currently enabled or disabled. *Actions* are usually defined using the class *AbstractAction*, an `abstract` class which includes a method

```
public void actionPerformed(ActionEvent evt)
```

that must be defined in any concrete subclass. Often, this is done in an anonymous inner class. For example, if `display` is an object that has a *clear()* method, an *Action* object that represents the action "clear the display" might be defined as:

```
Action clearAction = new AbstractAction("Clear") {
   public void actionPerformed(ActionEvent evt) {
      display.clear();
   }
};
```

The parameter, `"Clear"`, in the constructor of the *AbstractAction* is the name of the action. Other properties can be set by calling the method `putValue(key,value)`, which is part of the *Action* interface. For example,

```
clearAction.putValue(Action.SHORT_DESCRIPTION, "Clear the Display");
```

sets the `SHORT_DESCRIPTION` property of the action to have the value "Clear the Display". The `key` parameter in the `putValue()` method is usually given as one of several constants defined in the *Action* interface. As another example, you can change the name of an action by using `Action.NAME` as the `key` in the `putValue()` method.

Once you have an *Action*, you can use it in the constructor of a button. For example, using the action `clearAction` defined above, we can create the *JButton*

```
JButton clearButton = new JButton( clearAction );
```

The name of the action will be used as the text of the button, and some other properties of the button will be taken from properties of the action. For example, if the `SHORT_DESCRIPTION` property of the action has a value, then that value is used as the tooltip text for the button. (The tooltip text appears when the user hovers the mouse over the button.) Furthermore, when you change a property of the action, the corresponding property of the button will also be changed.

The *Action* interface defines a `setEnabled()` method that is used to enable and disable the action. The `clearAction` action can be enabled and disabled by calling `clearAction.setEnabled(true)` and `clearAction.setEnabled(false)`. When you do this, any button that has been created from the action is also enabled or disabled at the same time.

Now of course, the question is, **why** should you want to use *Actions* at all? One advantage is that using actions can help you to organize your code better. You can create separate objects that represent each of the actions that can be performed in your program. This represents a nice division of responsibility. Of course, you could do the same thing with individual *ActionListener* objects, but then you couldn't associate descriptions and other properties with the actions.

More important is the fact that *Actions* can also be used in other places in the Java API. You can use an *Action* to create a *JMenuItem* in the same way as for a *JButton*:

```
JMenuItem clearCommand = new JMenuItem( clearAction );
```

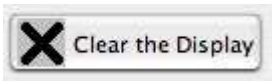
A *JMenuItem*, in fact, is a kind of button and shares many of the same properties that a *JButton* can have. You can use the **same** *Action* to create both a button and a menu item (or even several of each if you want). Whenever you enable or disable the action or change its name, the button and the menu item will **both** be changed to match. If you change the `NAME` property of the action, the text of both the menu item and the button will be set to the new name of the action. If you disable the action, both menu item and button will be disabled. You can think of the button and the menu items as being two presentations of the *Action*, and you don't have to keep track of the button or menu item after you create them. You can do everything that you need to do by manipulating the *Action* object.

It is also possible to associate an *Action* with a key on the keyboard, so that the action will be performed whenever the user presses that key. I won't explain how to do it here, but you can look up the documentation for the classes `javax.swing.InputMap` and `javax.swing.ActionMap`.

By the way, if you want to add a menu item that is defined by an *Action* to a menu, you don't even need to create the *JMenuItem* yourself. You can add the action object directly to the menu, and the menu item will be created from the properties of the action. For example, if `menu` is a *JMenu* and `clearAction` is an *Action*, you can simply say `menu.add(clearAction)`.

13.3.2 Icons on Buttons

In addition to—or instead of—text, buttons can also show icons. Icons are represented by the *Icon* interface and are usually created as *ImageIcons*, as discussed in Subsection 13.1.4. For example, here is a picture of a button that displays an image of a large "X" as its icon:



The icon for a button can be set by calling the button's `setIcon()` method, or by passing the icon object as a parameter to the constructor when the button is created. To create the button shown above, I created an *ImageIcon* from a *BufferedImage* on which I drew the picture that I wanted, and I constructed the *JButton* using a constructor that takes both the text and the icon for the button as parameters. Here's the code segment that does it:

```
BufferedImage image = new BufferedImage(24,24,BufferedImage.TYPE_INT_RGB);

Graphics2D g2 = (Graphics2D)image.getGraphics();
g2.setColor(Color.LIGHT_GRAY);             // Draw the image for the icon.
g2.fillRect(0,0,24,24);
g2.setStroke( new BasicStroke(3) );        //    Use thick lines.
g2.setColor(Color.BLACK);
g2.drawLine(4,4,20,20);                    //    Draw the "X".
g2.drawLine(4,20,20,4);
g2.dispose();

Icon clearIcon = new ImageIcon(image);     // Create the icon.

JButton clearButton = new JButton("Clear the Display", clearIcon);
```

You can create a button with an icon but no text by using a constructor that takes just the icon as parameter. Another alternative is for the button to get its icon from an *Action*. When a button is constructed from an action, it takes its icon from the value of the action property `Action.SMALL_ICON`. For example, suppose that we want to use an action named `clearAction` to create the button shown above. This could be done with:

```
clearAction.putValue( Action.SMALL_ICON, clearIcon );
JButton clearButton = new JButton( clearAction );
```

The icon could also be associated with the action by passing it as a parameter to the constructor of an *AbstractAction*:

```
Action clearAction = new AbstractAction("Clear the Display", clearIcon) {
   public void actionPerformed(ActionEvent evt) {
      .
      .  // Carry out the action.
      .
   }
}
JButton clearButton = new JButton( clearAction );
```

(In Java 6.0 and later, a button will use the value of the `Action.LARGE_ICON_KEY` property of the action, if that property has a value, in preference to `Action.SMALL_ICON`.)

The appearance of buttons can be tweaked in many ways. For example, you can change the size of the gap between the button's text and its icon. You can associate additional icons with a button that are used when the button is in certain states, such as when it is pressed or when it is disabled. It is even possible to change the positioning of the text with respect to the icon. For example, to place the text centered below the icon on a button, you can say:

```
button.setHorizontalTextPosition(JButton.CENTER);
button.setVerticalTextPosition(JButton.BOTTOM);
```

These methods and many others are defined in the class *AbstractButton*. This class is a superclass for *JMenuItem*, as well as for *JButton* and for the classes that define check boxes and radio buttons. Note in particular that an icon can be shown in a menu by associating the icon with a menu item or with the action that is used to create the menu item.

Finally, I will mention that it is possible to use icons on *JLabels* in much the same way that they can be used on *JButtons*. Placing an *ImageIcon* on a *JLabel* can be a convenient way to add a static image to your GUI.

13.3.3 Radio Buttons

The *JCheckBox* class was covered in Subsection 6.6.3, and the equivalent for use in menus, *JCheckBoxMenuItem*, in Subsection 6.8.1. A checkbox has two states, selected and not selected, and the user can change the state by clicking on the check box. The state of a checkbox can also be set programmatically by calling its `setSelected()` method, and the current value of the state can be checked using the `isSelected()` method.

Closely related to checkboxes are ***radio buttons***. Like a checkbox, a radio button can be either selected or not. However, radio buttons are expected to occur in groups, and at most one radio button in a group can be selected at any given time. In Java, a radio button is represented by an object of type *JRadioButton*. When used in isolation, a *JRadioButton* acts just like a *JCheckBox*, and it has the same methods and events. Ordinarily, however, a *JRadioButton* is used in a group. A group of radio buttons is represented by an object belonging to the class *ButtonGroup*. A *ButtonGroup* is **not** a component and does not itself have a visible representation on the screen. A *ButtonGroup* works behind the scenes to organize a group of radio buttons, to ensure that at most one button in the group can be selected at any given time.

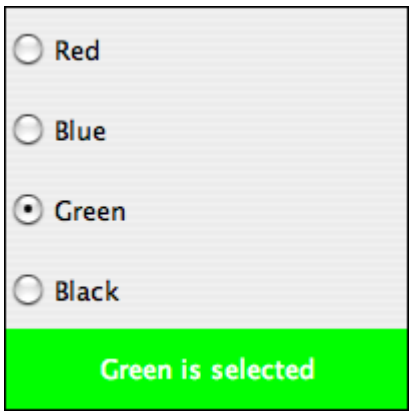
To use a group of radio buttons, you must create a *JRadioButton* object for each button in the group, and you must create one object of type *ButtonGroup* to organize the individual buttons into a group. Each *JRadioButton* must be added individually to some container, so that it will appear on the screen. (A *ButtonGroup* plays no role in the placement of the buttons on the screen.) Each *JRadioButton* must also be added to the *ButtonGroup*, which has an `add()` method for this purpose. If you want one of the buttons to be selected initially, you can call `setSelected(true)` for that button. If you don't do this, then none of the buttons will be selected until the user clicks on one of them.

As an example, here is how you could set up a set of radio buttons that can be used to select a color:

```
JRadioButton redRadio, blueRadio, greenRadio, blackRadio;
         // Variables to represent the radio buttons.
         // These should probably be instance variables, so
         // that they can be used throughout the program.

ButtonGroup colorGroup = new ButtonGroup();

redRadio = new JRadioButton("Red");  // Create a button.
colorGroup.add(redRadio);            // Add it to the group.

blueRadio = new JRadioButton("Blue");
colorGroup.add(blueRadio);

greenRadio = new JRadioButton("Green");
colorGroup.add(greenRadio);

blackRadio = new JRadioButton("Black");
colorGroup.add(blackRadio);

redRadio.setSelected(true);  // Make an initial selection.
```

The individual buttons must still be added to a container if they are to appear on the screen. If you want to respond immediately when the user clicks on one of the radio buttons, you can register an *ActionListener* for each button. Just as for checkboxes, it is not always necessary to register listeners for radio buttons. In some cases, you can simply check the state of each button when you need to know it, using the button's `isSelected()` method.

All this is demonstrated in the sample program *RadioButtonDemo.java*. The program shows four radio buttons. When the user selects one of the radio buttons, the text and background color of a label is changed. Here is a picture of the program, with the "Green" radio button selected:



You can add the equivalent of a group of radio buttons to a menu by using the class *JRadioButtonMenuItem*. To use this class, create several objects of this type, and create a *ButtonGroup* to manage them. Add each *JRadioButtonMenuItem* to the *ButtonGroup*, and also add them to a *JMenu*. If you want one of the items to be selected initially, call its `setSelected()` method to set its selection state to true. You can add *ActionListeners* to each *JRadioButtonMenuItem* if you need to take some action when the user selects the menu item; if not, you can simply check the selected states of the buttons whenever you need to know them. As an example, suppose that `menu` is a *JMenu*. Then you can add a group of buttons to `menu` as follows:

```
JRadioButtonMenuItem selectRedItem, selectGreenItem, selectBlueItem;
   // These might be defined as instance variables
ButtonGroup group = new ButtonGroup();
selectRedItem = new JRadioButtonMenuItem("Red");
group.add(selectRedItem);
menu.add(selectRedItem);
selectGreenItem = new JRadioButtonMenuItem("Green");
group.add(selectGreenItem);
menu.add(selectGreenItem);
selectBlueItem = new JRadioButtonMenuItem("Blue");
group.add(selectBlueItem);
menu.add(selectBlueItem);
```

* * *

When it's drawn on the screen, a *JCheckBox* includes a little box that is either checked or unchecked to show the state of the box. That box is actually a pair of *Icons*. One icon is shown when the check box is unselected; the other is shown when it is selected. You can change the appearance of the check box by substituting different icons for the standard ones.

The icon that is shown when the check box is unselected is just the main icon for the *JCheckBox*. You can provide a different unselected icon in the constructor or you can change the icon using the `setIcon()` method of the *JCheckBox* object. To change the icon that is shown when the check box is selected, use the `setSelectedIcon()` method of the *JCheckBox*. All this applies equally to *JRadioButton*, *JCheckBoxMenuItem*, and *JRadioButtonMenuItem*.

An example of this can be found in the sample program *ToolBarDemo.java*, which is discussed in the next subsection. That program creates a set of radio buttons that use custom icons. The buttons are created by the following method:

```
/**
 * Create a JRadioButton and add it to a specified button group.  The button
 * is meant for selecting a drawing color in the display.  The color is used to
 * create two custom icons, one for the unselected state of the button and one
 * for the selected state.  These icons are used instead of the usual
 * radio button icons.
 * @param c the color of the button, and the color to be used for drawing.
 *    (Note that c has to be "final" since it is used in the anonymous inner
 *    class that defines the response to ActionEvents on the button.)
 * @param grp the ButtonGroup to which the radio button will be added.
 * @param selected if true, then the state of the button is set to selected.
 * @return the radio button that was just created; sorry, but the button
      is not as pretty as I would like!
 */
private JRadioButton makeColorRadioButton(final Color c,
```

```
                                                ButtonGroup grp, boolean selected) {

    /* Create an ImageIcon for the normal, unselected state of the button,
       using a BufferedImage that is drawn here from scratch. */

    BufferedImage image = new BufferedImage(30,30,BufferedImage.TYPE_INT_RGB);
    Graphics g = image.getGraphics();
    g.setColor(Color.LIGHT_GRAY);
    g.fillRect(0,0,30,30);
    g.setColor(c);
    g.fill3DRect(1, 1, 24, 24, true);
    g.dispose();
    Icon unselectedIcon = new ImageIcon(image);

    /* Create an ImageIcon for the selected state of the button. */

    image = new BufferedImage(30,30,BufferedImage.TYPE_INT_RGB);
    g = image.getGraphics();
    g.setColor(Color.DARK_GRAY);
    g.fillRect(0,0,30,30);
    g.setColor(c);
    g.fill3DRect(3, 3, 24, 24, false);
    g.dispose();
    Icon selectedIcon = new ImageIcon(image);

    /* Create and configure the button. */

    JRadioButton button = new JRadioButton(unselectedIcon);
    button.setSelectedIcon(selectedIcon);
    button.addActionListener( new ActionListener() {
       public void actionPerformed(ActionEvent e) {
            // The action for this button sets the current drawing color
            // in the display to c.
          display.setCurrentColor(c);
       }
    });
    grp.add(button);
    if (selected)
       button.setSelected(true);

    return button;
} // end makeColorRadioButton
```

* * *

It is possible to create radio buttons and check boxes from *Actions*. The button takes its name, main icon, tooltip text, and enabled/disabled state from the action. In Java 5.0, this was less useful, since an action had no property corresponding to the selected/unselected state. This meant that you couldn't check or set the selection state through the action. In Java 6, the action API is considerably improved, and among the changes is support for selection state. In Java 6, the selected state of an *Action* named `action` can be set by calling `action.putValue(Action.SELECTED_KEY,true)` and `action.putValue(Action.SELECTED_KEY,false)`. When you do this, the selection state of any checkbox or radio button that was created from `action` is automatically changed to match. Conversely, when the state of the checkbox or radio button is changed in some other way, the property of the action—and hence of any other components created

from the action—will automatically change as well. The state can be checked by calling `action.getValue(Action.SELECTED_KEY)`.

13.3.4 Toolbars

It has become increasingly common for programs to have a row of small buttons along the top or side of the program window that offer access to some of the commonly used features of the program. The row of buttons is known as a ***tool bar***. Typically, the buttons in a tool bar are presented as small icons, with no text. Tool bars can also contain other components, such as *JTextFields* and *JLabels*.

In Swing, tool bars are represented by the class *JToolBar*. A *JToolBar* is a container that can hold other components. It is also itself a component, and so can be added to other containers. In general, the parent component of the tool bar should use a *BorderLayout*. The tool bar should occupy one of the edge positions—`NORTH`, `SOUTH`, `EAST`, or `WEST`—in the *BorderLayout*. Furthermore, the other three edge positions should be empty. The reason for this is that it might be possible (depending on the platform and configuration) for the user to drag the tool bar from one edge position in the parent container to another. It might even be possible for the user to drag the tool bar off its parent entirely, so that it becomes a separate window.

Here is a picture of a tool bar from the sample program *ToolBarDemo.java*.

In this program, the user can draw colored curves in a large drawing area. The first three buttons in the tool bar are a set of radio buttons that control the drawing color. The fourth button is a push button that the user can click to clear the drawing.

Tool bars are easy to use. You just have to create the *JToolBar* object, add it to a container, and add some buttons and possibly other components to the tool bar. One fine point is adding space to a tool bar, such as the gap between the radio buttons and the push button in the sample program. You can leave a gap by adding a separator to the tool bar. For example:

```
toolbar.addSeparator(new Dimension(20,20));
```

This adds an invisible 20-by-20 pixel block to the tool bar. This will appear as a 20 pixel gap between components.

Here is the constructor from the `ToolBarDemo` program. It shows how to create the tool bar and place it in a container. Note that class *ToolBarDemo* is a subclass of *JPanel*, and the tool bar and display are added to the panel object that is being constructed:

```
public ToolBarDemo() {

   setLayout(new BorderLayout(2,2));
   setBackground(Color.GRAY);
   setBorder(BorderFactory.createLineBorder(Color.GRAY,2));

   display = new Display();
   add(display, BorderLayout.CENTER);

   JToolBar toolbar = new JToolBar();
   add(toolbar, BorderLayout.NORTH);

   ButtonGroup group = new ButtonGroup();
```

```
        toolbar.add( makeColorRadioButton(Color.RED,group,true) );
        toolbar.add( makeColorRadioButton(Color.GREEN,group,false) );
        toolbar.add( makeColorRadioButton(Color.BLUE,group,false) );
        toolbar.addSeparator(new Dimension(20,20));

        toolbar.add( makeClearButton() );

    }
```

Note that the gray outline of the tool bar comes from two sources: The line at the bottom shows the background color of the main panel, which is visible because the *BorderLayout* that is used on that panel has vertical and horizontal gaps of 2 pixels. The other three sides are part of the border of the main panel.

If you want a vertical tool bar that can be placed in the `EAST` or `WEST` position of a *BorderLayout*, you should specify the orientation in the tool bar's constructor:

```
JToolBar toolbar = new JToolBar( JToolBar.VERTICAL );
```

The default orientation is `JToolBar.HORIZONTAL`. The orientation is adjusted automatically when the user drags the tool bar into a new position. If you want to prevent the user from dragging the tool bar, just say `toolbar.setFloatable(false)`.

13.3.5 Keyboard Accelerators

In most programs, commonly used menu commands have keyboard equivalents. The user can type the keyboard equivalent instead of selecting the command from the menu, and the result will be exactly the same. Typically, for example, the "Save" command has keyboard equivalent `CONTROL-S`, and the "Undo" command corresponds to `CONTROL-Z`. (Under Mac OS, the keyboard equivalents for these commands would probably be `META-S` and `META-Z`, where `META` refers to holding down the "apple" key.) The keyboard equivalents for menu commands are referred to as ***accelerators***.

The class `javax.swing.KeyStroke` is used to represent key strokes that the user can type on the keyboard. A key stroke consists of pressing a key, possibly while holding down one or more of the modifier keys `control`, `shift`, `alt`, and `meta`. The *KeyStroke* class has a static method, `getKeyStroke(String)`, that makes it easy to create key stroke objects. For example,

```
KeyStroke.getKeyStroke( "ctrl S" )
```

returns a *KeyStroke* that represents the action of pressing the "S" key while holding down the control key. In addition to "ctrl", you can use the modifiers "shift", "alt", and "meta" in the string that describes the key stroke. You can even combine several modifiers, so that

```
KeyStroke.getKeyStroke( "ctrl shift Z" )
```

represents the action of pressing the "Z" key while holding down both the control and the shift keys. When the key stroke involves pressing a character key, the character must appear in the string in upper case form. You can also have key strokes that correspond to non-character keys. The number keys can be referred to as "1", "2", etc., while certain special keys have names such as "F1", "ENTER", and "LEFT" (for the left arrow key). The class *KeyEvent* defines many constants such as `VK_ENTER`, `VK_LEFT`, and `VK_S`. The names that are used for keys in the keystroke description are just these constants with the leading "`VK_`" removed.

There are at least two ways to associate a keyboard accelerator with a menu item. One is to use the `setAccelerator()` method of the menu item object:

```
JMenuItem saveCommand = new JMenuItem( "Save..." );
saveCommand.setAccelerator( KeyStroke.getKeyStroke("ctrl S") );
```

The other technique can be used if the menu item is created from an *Action*. The action property `Action.ACCELERATOR_KEY` can be used to associate a *KeyStroke* with an *Action*. When a menu item is created from the action, the keyboard accelerator for the menu item is taken from the value of this property. For example, if `redoAction` is an *Action* representing a "Redo" action, then you might say:

```
redoAction.putValue( Action.ACCELERATOR_KEY,
                              KeyStroke.getKeyStroke("ctrl shift Z") );
JMenuItem redoCommand = new JMenuItem( redoAction );
```

or, alternatively, you could simply add the action to a *JMenu*, `editMenu`, with `editMenu.add(redoAction)`. (Note, by the way, that accelerators apply only to menu items, not to push buttons. When you create a *JButton* from an action, the `ACCELERATOR_KEY` property of the action is ignored.)

Note that you can use accelerators for *JCheckBoxMenuItems* and *JRadioButtonMenuItems*, as well as for simple *JMenuItems*.

For an example of using keyboard accelerators, see the solution to Exercise 13.2.

* * *

By the way, as noted above, in the Mac OS operating system, the meta (or apple) key is usually used for keyboard accelerators instead of the control key. If you would like to make your program more Mac-friendly, you can test whether your program is running under Mac OS and, if so, adapt your accelerators to the Mac OS style. The recommended way to detect Mac OS is to test the value of `System.getProperty("mrj.version")`. This function call happens to return a non-null value under Mac OS but returns null under other operating systems. For example, here is a simple utility routine for making Mac-friendly accelerators:

```
/**
 * Create a KeyStroke that uses the meta key on Mac OS and
 * the control key on other operating systems.
 * @param description a string that describes the keystroke,
 *   without the "meta" or "ctrl"; for example, "S" or
 *   "shift Z" or "alt F1"
 * @return a keystroke created from the description string
 *   with either "ctrl " or "meta " prepended
 */
private static KeyStroke makeAccelerator(String description) {
   String commandKey;
   if ( System.getProperty("mrj.version") == null )
      commandKey = "ctrl";
   else
      commandKey = "meta";
   return KeyStroke.getKeyStroke( commandKey + " " + description );
}
```

13.3.6 HTML on Buttons

As a final stop in this brief tour of ways to spiff up your buttons, I'll mention the fact that the text that is displayed on a button can be specified in HTML format. HTML is the markup language that is used to write web pages. A brief introduction to HTML can be found in

Subsection 6.2.3. HTML allows you to apply color or italics or other styles to just part of the text on your buttons. It also makes it possible to have buttons that display multiple lines of text. (You can also use HTML on *JLabels*, which can be even more useful.) Here's a picture of a button with HTML text (along with a "Java" icon):

If the string of text that is applied to a button starts with "`<html>`", then the string is interpreted as HTML. The string does not have to use strict HTML format; for example, you don't need a closing `</html>` at the end of the string. To get multi-line text, use `<br>` in the string to represent line breaks. If you would like the lines of text to be center justified, include the entire text (except for the `<html>`) between `<center>` and `</center>`. For example,

```
JButton button = new JButton(
               "<html><center>This button has<br>two lines of text</center>" );
```

creates a button that displays two centered lines of text. You can apply italics to part of the string by enclosing that part between `<i>` and `</i>`. Similarly, use `<b>...</b>` for bold text and `<u>...</u>` for underlined text. For green text, enclose the text between `<font color=green>` and `</font >`. You can, of course, use other colors in place of "green." The "Java" button that is shown above was created using:

```
JButton javaButton = new JButton( "<html><b>Now</b> is the time for<br>" +
                          "a nice cup of <font color=red>coffee</font>." );
```

Other HTML features can also be used on buttons and labels—experiment to see what you can get away with!

13.4 Complex Components and MVC

Since even buttons turn out to be pretty complex, as seen in the previous section, you might guess that there is a lot more complexity lurking in the Swing API. While this is true, a lot of that complexity works to your benefit as a programmer, since a lot of it is hidden in normal uses of Swing components. For example, you don't have to know about all the complex details of buttons in order to use them effectively in most programs.

Swing defines several component classes that are much more complex than those we have looked at so far, but even the most complex components are not very difficult to use for many purposes. In this section, we'll look at components that support display and manipulation of lists, tables, and text documents. To use these complex components effectively, you'll need to know something about the Model-View-Controller pattern that is used as a basis for the design of many Swing components. This pattern is discussed in the first part of this section.

This section is our last look at Swing components, but there are a number of component classes that have not even been touched on in this book. Some useful ones that you might want to look into include: *JTabbedPane*, *JSplitPane*, *JTree*, *JSpinner*, *JPopupMenu*, *JProgressBar*, and *JScrollBar*.

At the end of the section, we'll look briefly at the idea of writing custom component classes—something that you might consider when even the large variety of components that are already defined in Swing don't do quite what you want.

13.4.1 Model-View-Controller

One of the principles of object-oriented design is division of responsibilities. Ideally, an object should have a single, clearly defined role, with a limited realm of responsibility. One application of this principle to the design of graphical user interfaces is the ***MVC pattern***. "MVC" stands for "Model-View-Controller" and refers to three different realms of responsibility in the design of a graphical user interface.

When the MVC pattern is applied to a component, the ***model*** consists of the data that represents the current state of the component. The ***view*** is simply the visual presentation of the component on the screen. And the ***controller*** is the aspect of the component that carries out actions in response to events generated by the user (or by other sources such as timers). The idea is to assign responsibility for the model, the view, and the controller to different objects.

The view is the easiest part of the MVC pattern to understand. It is often represented by the component object itself, and its responsibility is to draw the component on the screen. In doing this, of course, it has to consult the model, since the model represents the state of the component, and that state can determine what appears on the screen. To get at the model data—which is stored in a separate object according to the MVC pattern—the component object needs to keep a reference to the model object. Furthermore, when the model changes, the view might have to be redrawn to reflect the changed state. The component needs some way of knowing when changes in the model occur. Typically, in Java, this is done with events and listeners. The model object is set up to generate events when its data changes. The view object registers itself as a listener for those events. When the model changes, an event is generated, the view is notified of that event, and the view responds by updating its appearance on the screen.

When MVC is used for Swing components, the controller is generally not so well defined as the model and view, and its responsibilities are often split among several objects. The controller might include mouse and keyboard listeners that respond to user events on the view; *Actions* that respond to menu commands or buttons; and listeners for other high-level events, such as those from a slider, that affect the state of the component. Usually, the controller responds to events by making modifications to the model, and the view is changed only indirectly, in response to the changes in the model.

The MVC pattern is used in many places in the design of Swing. It is even used for buttons. The state of a Swing button is stored in an object of type *ButtonModel*. The model stores such information as whether the button is enabled, whether it is selected, and what *ButtonGroup* it is part of, if any. If `button` is of type *JButton* (or one of the other subclasses of *AbstractButton*), then its *ButtonModel* can be obtained by calling `button.getModel()`. In the case of buttons, you might never need to use the model or even know that it exists. But for the list and table components that we will look at next, knowledge of the model is essential.

13.4.2 Lists and ListModels

A *JList* is a component that represents a list of items that can be selected by the user. The sample program *SillyStamper.java* allows the user to select one icon from a *JList* of *Icons*. The user selects an icon from the list by clicking on it. The selected icon can be "stamped" onto a drawing area by clicking on the drawing area. (The icons in this program are from the KDE desktop project.) Here is a picture of the program with several icons already stamped onto the drawing area and with the "light bulb" icon selected:

Note that the scrollbar in this program is not part of the *JList*. To add a scrollbar to a list, the list must be placed into a *JScrollPane*. See Subsection 6.6.4, where the use of *JScrollPane* to hold a *JTextArea* was discussed. Scroll panes are used in the same way with lists and with other components. In this case, the *JList*, `iconList`, was added to a scroll pane and the scroll pane was added to a panel with the single command:

```
add( new JScrollPane(iconList), BorderLayout.EAST );
```

One way to construct a *JList* is from an array that contains the objects that will appear in the list. The items can be of any type, but only icons and strings can actually appear in the list; an item that is not of type *Icon* or *String* is converted into a string by calling its `toString()` method. (It's possible to "teach" a *JList* to display other types of items; see the `setCellRenderer()` method in the *JList* class.) In the `SillyStamper` program, the images for the icons are read from resource files, the icons are placed into an array, and the array is used to construct the list. This is done by the following method:

```
private JList createIconList() {

   String[] iconNames = new String[] {
      "icon5.png", "icon7.png", "icon8.png", "icon9.png", "icon10.png",
      "icon11.png", "icon24.png", "icon25.png", "icon26.png", "icon31.png",
      "icon33.png", "icon34.png"
   };              // Array containing resource file names for the icon images.

   iconImages = new Image[iconNames.length];

   ClassLoader classLoader = getClass().getClassLoader();
   Toolkit toolkit = Toolkit.getDefaultToolkit();
   try {                          // Get the icon images from the resource files.
      for (int i = 0; i < iconNames.length; i++) {
         URL imageURL = classLoader.getResource("stamper_icons/" + iconNames[i]);
         if (imageURL == null)
            throw new Exception();
         iconImages[i] = toolkit.createImage(imageURL);
      }
   }
   catch (Exception e) {
      iconImages = null;
      return null;
```

```
        }

        ImageIcon[] icons = new ImageIcon[iconImages.length];
        for (int i = 0; i < iconImages.length; i++)          // Create the icons.
           icons[i] = new ImageIcon(iconImages[i]);

        JList list = new JList(icons);         // A list containing the image icons.
        list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        list.setSelectedIndex(0);   // First item in the list is currently selected.

        return list;
    }
```

By default, the user can select any number of items in a list. A single item is selected by clicking on it. Multiple items can be selected by shift-clicking and by either control-clicking or meta-clicking (depending on the platform). In the `SillyStamper` program, I wanted to restrict the selection so that only one item can be selected at a time. This restriction is imposed by calling

```
list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
```

With this selection mode, when the user selects an item, the previously selected item, if any, is deselected. Note that the selection can be changed by the program by calling `list.setSelectedIndex(itemNum)`. Items are numbered starting from zero. To find out the currently selected item in single selection mode, call `list.getSelectedIndex()`. This returns the item number of the selected item, or `-1` if no item is currently selected. If multiple selections are allowed, you can call `list.getSelectedIndices()`, which returns an array of **ints** that contains the item numbers of all selected items.

Now, the list that you see on the screen is only the *view* aspect of the list. The *controller* consists of the listener objects that respond when the user clicks an item in the list. For its *model*, a *JList* uses an object of type *ListModel*. This is the object that knows the actual list of items. Now, a model is defined not only by the data that it contains but by the set of operations that can be performed on the data. When a *JList* is constructed from an array of objects, the model that is used is very simple. The model can tell you how many items it contains and what those items are, but it can't do much else. In particular, there is no way to add items to the list or to delete items from the list! If you need that capability, you will have to use a different list model.

The class *DefaultListModel* defines list models that support adding items to and removing items from the list. (Note that the list model that you get when you create a *JList* from an array is **not** of this type.) If `dlmodel` is of type *DefaultListModel*, the following methods, among others, are defined:

- `dlmodel.getSize()` — returns the number of items.
- `dlmodel.getElementAt(index)` — returns the item at position `index` in the list.
- `dlmodel.addElement(item)` — Adds `item` to the end of the list; `item` can be any *Object*.
- `dlmodel.insertElementAt(item, index)` — inserts the specified `item` into the list at the specified `index`; items that come after that position in the list are moved down to make room for the new item.
- `dlmodel.setElementAt(item, index)` — Replaces the item that is currently at position `index` in the list with `item`.
- `dlmodel.remove(index)` — removes the item at position `index` in the list.

- `dlmodel.removeAllElements()` — removes everything from the list, leaving it empty.

To use a modifiable *JList*, you should create a *DefaultListModel*, add any items to it that should be in the list initially, and pass it to the *JList* constructor. For example:

```
DefaultListModel listModel;  // Should probably be instance variables!
JList flavorList;

listModel = new DefaultListModel();    // Create the model object.

listModel.addElement("Chocolate");     // Add items to the model.
listModel.addElement("Vanilla");
listModel.addElement("Strawberry");
listModel.addElement("Rum Raisin");

flavorList = new JList(listModel);     // Create the list component.
```

By keeping a reference to the model around in an instance variable, you will be able to add and delete flavors as the program is running by calling the appropriate methods in `listModel`. Keep in mind that changes that are made to the *model* will automatically be reflected in the *view*. Behind the scenes, when a list model is modified, it generates an event of type *ListDataEvent*. The *JList* registers itself with its model as a listener for these events, and it responds to an event by redrawing itself to reflect the changes in the model. The programmer doesn't have to take any extra action, beyond changing the model.

By the way, the model for a *JList* actually has another part in addition to the *ListModel*: An object of type *ListSelectionModel* stores information about which items in the list are currently selected. When the model is complex, it's not uncommon to use several model objects to store different aspects of the state.

13.4.3 Tables and TableModels

Like a *JList*, a *JTable* displays a collection of items to the user. However, tables are much more complicated than lists. Perhaps the most important difference is that it is possible for the user to edit items in the table. Table items are arranged in a grid of rows and columns. Each grid position is called a ***cell*** of the table. Each column can have a ***header***, which appears at the top of the column and contains a name for the column.

It is easy to create a *JTable* from an array that contains the names of the columns and a two-dimensional array that contains the items that go into the cells of the table. As an example, the sample program *StatesAndCapitalsTableDemo.java* creates a table with two columns named "State" and "Capital City." The first column contains a list of the states of the United States and the second column contains the name of the capital city of each state. The table can be created as follows:

```
String[][] statesAndCapitals = new String[][] {
           { "Alabama", "Montgomery" },
           { "Alaska", "Juneau" },
           { "Arizona", "Phoenix" },
                .
                .
                .
           { "Wisconsin", "Madison" },
           { "Wyoming", "Cheyenne" }
        };
```

```
String[] columnHeads = new String[] { "State", "Capital City" };

JTable table = new JTable(statesAndCapitals, columnHeads);
```

Since a table does not come with its own scroll bars, it is almost always placed in a *JScrollPane* to make it possible to scroll the table. In the example program this is done with:

```
add( new JScrollPane(table), BorderLayout.CENTER );
```

The column headers of a *JTable* are not actually part of the table; they are in a separate component. But when you add the table to a *JScrollPane*, the column headers are automatically placed at the top of the pane.

Using the default settings, the user can edit any cell in the table. (To select an item for editing, click it and start typing. The arrow keys can be used to move from one cell to another.) The user can change the order of the columns by dragging a column header to a new position. The user can also change the width of the columns by dragging the line that separates neighboring column headers. You can try all this in the sample program; there is an applet version in the on-line version of this section.

Allowing the user to edit all entries in the table is not always appropriate; certainly it's not appropriate in the "states and capitals" example. A *JTable* uses an object of type *TableModel* to store information about the contents of the table. The model object is also responsible for deciding whether or not the user should be able to edit any given cell in the table. *TableModel* includes the method

```
public boolean isCellEditable(int rowNum, columnNum)
```

where `rowNum` and `columnNum` are the position of a cell in the grid of rows and columns that make up the table. When the controller wants to know whether a certain cell is editable, it calls this method in the table model. If the return value is true, the user is allowed to edit the cell.

The default model that is used when the table is created, as above, from an array of objects allows editing of all cells. For this model, the return value of `isCellEditable()` is true in all cases. To make some cells non-editable, you have to provide a different model for the table. One way to do this is to create a subclass of *DefaultTableModel* and override the `isCellEditable()` method. (*DefaultTableModel* and some other classes that are discussed in this section are defined in the package `javax.swing.table`.) Here is how this might be done in the "states and capitals" program to make all cells non-editable:

```
TableModel model = new DefaultTableModel(statesAndCapitals,columnHeads) {
   public boolean isCellEditable(int row, int col) {
      return false;
   }
};
JTable table = new JTable(model);
```

Here, an anonymous subclass of *DefaultTableModel* is created in which the `isCellEditable()` method returns `false` in all cases, and the model object that is created from that class is passed as a parameter to the *JTable* constructor.

The *DefaultTableModel* class defines many methods that can be used to modify the table, including for example: `setValueAt(item,rowNum,colNum)` to change the item in a given cell; `removeRow(rowNum)` to delete a row; and `addRow(itemArray)` to add a new row at the end of the table that contains items from the array `itemArray`. Note that if the item in a given cell

is `null`, then that cell will be empty. Remember, again, that when you modify the model, the view is automatically updated to reflect the changes.

In addition to the `isCellEditable()` method, the table model method that you are most likely to want to override is `getColumnClass()`, which is defined as

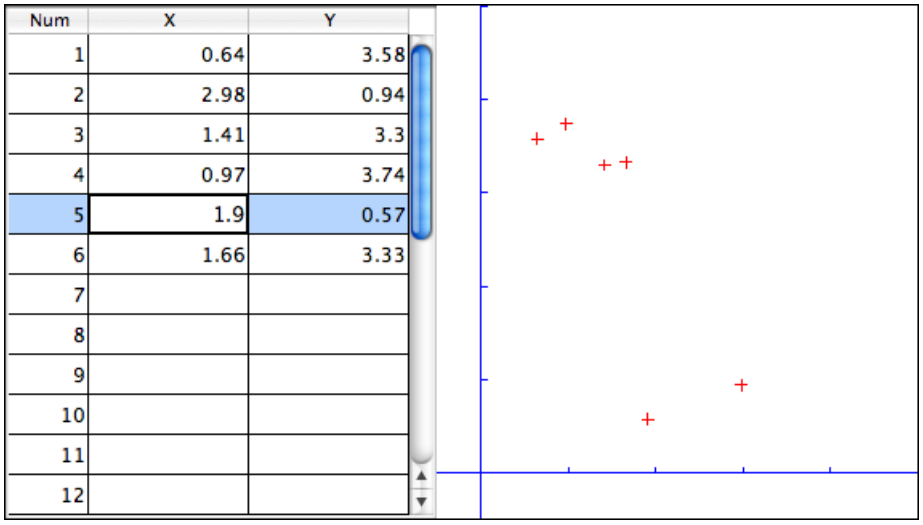
```
public Class<?> getColumnClass(columnNum)
```

The purpose of this method is to specify what kind of values are allowed in the specified column. The return value from this method is of type *Class*. (The "`<?>`" is there for technical reasons having to do with generic programming. See Section 10.5, but don't worry about understanding it here.) Although class objects have crept into this book in a few places—in the discussion of *ClassLoaders* in Subsection 13.1.3 for example—this is the first time we have directly encountered the class named *Class*. An object of type *Class* represents a class. A *Class* object is usually obtained from the name of the class using expressions of the form "`Double.class`" or "`JTable.class`". If you want a three-column table in which the column types are *String*, *Double*, and *Boolean*, you can use a table model in which `getColumnClass` is defined as:

```
public Class<?> getColumnClass(columnNum) {
   if (columnNum == 0)
      return String.class;
   else if (columnNum = 1)
      return Double.class;
   else
      return Boolean.class;
}
```

The table will call this method and use the return value to decide how to display and edit items in the table. For example, if a column is specified to hold *Boolean* values, the cells in that column will be displayed and edited as check boxes. For numeric types, the table will not accept illegal input when the user types in the value. (It is possible to change the way that a table edits or displays items. See the methods `setDefaultEditor()` and `setDefaultRenderer()` in the *JTable* class.)

As an alternative to using a subclass of *DefaultTableModel*, a custom table model can also be defined using a subclass of *AbstractTableModel*. Whereas *DefaultTableModel* provides a lot of predefined functionality, *AbstractTableModel* provides very little. However, using *AbstractTableModel* gives you the freedom to represent the table data any way you want. The sample program *ScatterPlotTableDemo.java* uses a subclass of *AbstractTableModel* to define the model for a *JTable*. In this program, the table has three columns. The first column holds a row number and is not editable. The other columns hold values of type *Double*; these two columns represent the x- and y-coordinates of points in the plane. The points themselves are graphed in a "scatter plot" next to the table. Initially, the program fills in the first six points with random values. Here is a picture of the program, with the x-coordinate in row 5 selected for editing:



Note, by the way, that in this program, the scatter plot can be considered to be a view of the table model, in the same way that the table itself is. The scatter plot registers itself as a listener with the model, so that it will receive notification whenever the model changes. When that happens, the scatter plot redraws itself to reflect the new state of the model. It is an important property of the MVC pattern that several views can share the same model, offering alternative presentations of the same data. The views don't have to know about each other or communicate with each other except by sharing the model. Although I didn't do it in this program, it would even be possible to add a controller to the scatter plot view. This would let the user drag a point in the scatter plot to change its coordinates. Since the scatter plot and table share the same model, the values in the table would automatically change to match.

Here is the definition of the class that defines the model in the scatter plot program. All the methods in this class must be defined in any subclass of *AbstractTableModel* except for `setValueAt()`, which only has to be defined if the table is modifiable.

```
/**
 * This class defines the TableModel that is used for the JTable in this
 * program.  The table has three columns.  Column 0 simply holds the
 * row number of each row.  Column 1 holds the x-coordinates of the
 * points for the scatter plot, and Column 2 holds the y-coordinates.
 * The table has 25 rows.  No support is provided for adding more rows.
 */
private class CoordInputTableModel extends AbstractTableModel {

   private Double[] xCoord = new Double[25];  // Data for Column 1.
   private Double[] yCoord = new Double[25];  // Data for Column 2.
        // Initially, all the values in the array are null, which means
        // that all the cells are empty.

   public int getColumnCount() {  // Tells caller how many columns there are.
      return 3;
   }

   public int getRowCount() {  // Tells caller how many rows there are.
      return xCoord.length;
   }
```

```
        public Object getValueAt(int row, int col) {  // Get value from cell.
           if (col == 0)
              return (row+1);        // Column 0 holds the row number.
           else if (col == 1)
              return xCoord[row];    // Column 1 holds the x-coordinates.
           else
              return yCoord[row];    // Column 2 holds the y-coordinates.
        }

        public Class<?> getColumnClass(int col) {  // Get data type of column.
           if (col == 0)
              return Integer.class;
           else
              return Double.class;
        }

        public String getColumnName(int col) {  // Returns a name for column header.
           if (col == 0)
              return "Num";
           else if (col == 1)
              return "X";
           else
              return "Y";
        }

        public boolean isCellEditable(int row, int col) { // Can user edit cell?
           return col > 0;
        }

        public void setValueAt(Object obj, int row, int col) {
              // (This method is called by the system if the value of the cell
              // needs to be changed because the user has edited the cell.
              // It can also be called to change the value programmatically.
              // In this case, only columns 1 and 2 can be modified, and the data
              // type for obj must be Double.  The method fireTableCellUpdated()
              // has to be called to send an event to registered listeners to
              // notify them of the modification to the table model.)
           if (col == 1)
              xCoord[row] = (Double)obj;
           else if (col == 2)
              yCoord[row] = (Double)obj;
           fireTableCellUpdated(row, col);
        }

     }  // end nested class CoordInputTableModel
```

In addition to defining a custom table model, I customized the appearance of the table in several ways. Because this involves changes to the view, most of the changes are made by calling methods in the *JTable* object. For example, since the default height of the cells was too small for my taste, I called `table.setRowHeight(25)` to increase the height. To make lines appear between the rows and columns, I found that I had to call both `table.setShowGrid(true)` and `table.setGridColor(Color.BLACK)`. Some of the customization has to be done to other objects. For example, to prevent the user from changing the order of the columns by dragging the column headers, I had to use

```
table.getTableHeader().setReorderingAllowed(false);
```

Tables are quite complex, and I have only discussed a part of the table API here. Nevertheless, I hope that you have learned enough to start using them and to learn more about them on your own.

13.4.4 Documents and Editors

As a final example of complex components, we look briefly at *JTextComponent* and its subclasses. A *JTextComponent* displays text that can, optionally, be edited by the user. Two subclasses, *JTextField* and *JTextArea*, were introduced in Subsection 6.6.4. But the real complexity comes in another subclass, *JEditorPane*, that supports display and editing of styled text. This allows features such as boldface and italic. A *JEditorPane* can even work with basic HTML documents.

It is almost absurdly easy to write a simple web browser program using a *JEditorPane*. This is done in the sample program *SimpleWebBrowser.java*. In this program, the user enters the URL of a web page, and the program tries to load and display the web page at that location. A *JEditorPane* can handle pages with content type "text/plain", "text/html", and "text/rtf". (The content type "text/rtf" represents styled or "rich text format" text. URLs and content types were covered in Subsection 11.4.1.) If `editPane` is of type *JEditorPane* and `url` is of type *URL*, then the statement "`editPane.setPage(url);`" is sufficient to load the page and display it. Since this can generate an exception, the following method is used in *SimpleWebBrowser.java* to display a page:

```
private void loadURL(URL url) {
   try {
      editPane.setPage(url);
   }
   catch (Exception e) {
      editPane.setContentType("text/plain"); // Set pane to display plain text.
      editPane.setText( "Sorry, the requested document was not found\n"
            +"or cannot be displayed.\n\nError:" + e);
   }
}
```

An HTML document can include links to other pages. When the user clicks on a link, the web browser should go to the linked page. A *JEditorPane* does not do this automatically, but it does generate an event of type *HyperLinkEvent* when the user clicks a link (provided that the edit pane has been set to be non-editable by the user). A program can register a listener for such events and respond by loading the new page.

There are a lot of web pages that a *JEditorPane* won't be able to display correctly, but it can be very useful in cases where you have control over the pages that will be displayed. A nice application is to distribute HTML-format help and information files with a program. The files can be stored as resource files in the jar file of the program, and a URL for a resource file can be obtained in the usual way, using the `getResource()` method of a *ClassLoader*. (See Subsection 13.1.3.)

It turns out, by the way, that *SimpleWebBrowser.java* is a little too simple. A modified version, *SimpleWebBrowserWithThread.java*, improves on the original by using a thread to load a page and by checking the content type of a page before trying to load it. It actually does work as a simple web browser.

The model for a *JTextComponent* is an object of type *Document*. If you want to be notified of changes in the model, you can add a listener to the model using

```
textComponent.getDocument().addDocumentListener(listener)
```

where `textComponent` is of type *JTextComponent* and `listener` is of type *DocumentListener*. The *Document* class also has methods that make it easy to read a document from a file and write a document to a file. I won't discuss all the things you can do with text components here. For one more peek at their capabilities, see the sample program *SimpleRTFEdit.java*, a very minimal editor for files that contain styled text of type "text/rtf."

13.4.5 Custom Components

Java's standard component classes are usually all you need to construct a user interface. At some point, however, you might need a component that Java doesn't provide. In that case, you can write your own component class, building on one of the components that Java does provide. We've already done this, actually, every time we've written a subclass of the *JPanel* class to use as a drawing surface. A *JPanel* is a blank slate. By defining a subclass, you can make it show any picture you like, and you can program it to respond in any way to mouse and keyboard events. Sometimes, if you are lucky, you don't need such freedom, and you can build on one of Java's more sophisticated component classes.

For example, suppose I have a need for a "stopwatch" component. When the user clicks on the stopwatch, I want it to start timing. When the user clicks again, I want it to display the elapsed time since the first click. The textual display can be done with a *JLabel*, but we want a *JLabel* that can respond to mouse clicks. We can get this behavior by defining a *StopWatchLabel* component as a subclass of the *JLabel* class. A *StopWatchLabel* object will listen for mouse clicks on itself. The first time the user clicks, it will change its display to "Timing..." and remember the time when the click occurred. When the user clicks again, it will check the time again, and it will compute and display the elapsed time. (Of course, I don't necessarily have to define a subclass. I could use a regular label in my program, set up a listener to respond to mouse events on the label, and let the program do the work of keeping track of the time and changing the text displayed on the label. However, by writing a new class, I have something that can be **reused** in other projects. I also have all the code involved in the stopwatch function collected together neatly in one place. For more complicated components, both of these considerations are very important.)

The *StopWatchLabel* class is not very hard to write. I need an instance variable to record the time when the user starts the stopwatch. Times in Java are measured in milliseconds and are stored in variables of type **long** (to allow for very large values). In the `mousePressed()` method, I need to know whether the timer is being started or stopped, so I need a **boolean** instance variable, `running`, to keep track of this aspect of the component's state. There is one more item of interest: How do I know what time the mouse was clicked? The method `System.currentTimeMillis()` returns the current time. But there can be some delay between the time the user clicks the mouse and the time when the `mousePressed()` routine is called. To make my stopwatch as accurate as possible, I don't want to know the current time. I want to know the exact time when the mouse was pressed. When I wrote the *StopWatchLabel* class, this need sent me on a search in the Java documentation. I found that if `evt` is an object of type *MouseEvent*, then the function `evt.getWhen()` returns the time when the event occurred. I call this function in the `mousePressed()` routine to determine the exact time when the user clicked on the label. The complete `StopWatch` class is rather short:

```
import java.awt.event.*;
import javax.swing.*;
```

```
/**
 * A custom component that acts as a simple stop-watch.  When the user clicks
 * on it, this component starts timing.  When the user clicks again,
 * it displays the time between the two clicks.  Clicking a third time
 * starts another timer, etc.  While it is timing, the label just
 * displays the message "Timing....".
 */
public class StopWatchLabel extends JLabel implements MouseListener {

   private long startTime;   // Start time of timer.
                             //   (Time is measured in milliseconds.)

   private boolean running;  // True when the timer is running.

   /**
    * Constructor sets initial text on the label to
    * "Click to start timer." and sets up a mouse listener
    * so the label can respond to clicks.
    */
   public StopWatchLabel() {
      super("  Click to start timer.  ", JLabel.CENTER);
      addMouseListener(this);
   }

   /**
    * Tells whether the timer is currently running.
    */
   public boolean isRunning() {
      return running;
   }

   /**
    * React when the user presses the mouse by starting or stopping
    * the timer and changing the text that is shown on the label.
    */
   public void mousePressed(MouseEvent evt) {
      if (running == false) {
            // Record the time and start the timer.
         running = true;
         startTime = evt.getWhen();  // Time when mouse was clicked.
         setText("Timing....");
      }
      else {
            // Stop the timer.  Compute the elapsed time since the
            // timer was started and display it.
         running = false;
         long endTime = evt.getWhen();
         double seconds = (endTime - startTime) / 1000.0;
         setText("Time: " + seconds + " sec.");
      }
   }

   public void mouseReleased(MouseEvent evt) { }
   public void mouseClicked(MouseEvent evt) { }
   public void mouseEntered(MouseEvent evt) { }
```

```
   public void mouseExited(MouseEvent evt) { }

}
```

Don't forget that since *StopWatchLabel* is a subclass of *JLabel*, you can do anything with a *StopWatchLabel* that you can do with a *JLabel*. You can add it to a container. You can set its font, foreground color, and background color. You can set the text that it displays (although this would interfere with its stopwatch function). You can even add a *Border* if you want.

Let's look at one more example of defining a custom component. Suppose that—for no good reason whatsoever—I want a component that acts like a *JLabel* except that it displays its text in mirror-reversed form. Since no standard component does anything like this, the *MirrorText* class is defined as a subclass of *JPanel*. It has a constructor that specifies the text to be displayed and a `setText()` method that changes the displayed text. The `paintComponent()` method draws the text mirror-reversed, in the center of the component. This uses techniques discussed in Subsection 13.1.1 and Subsection 13.2.1. Information from a *FontMetrics* object is used to center the text in the component. The reversal is achieved by using an off-screen canvas. The text is drawn to the off-screen canvas, in the usual way. Then the image is copied to the screen with the following command, where `OSC` is the variable that refers to the off-screen canvas, and `width` and `height` give the size of both the component and the off-screen canvas:

```
g.drawImage(OSC, width, 0, 0, height, 0, 0, width, height, this);
```

This is the version of `drawImage()` that specifies corners of destination and source rectangles. The corner `(0,0)` in `OSC` is matched to the corner `(width,0)` on the screen, while `(width,height)` is matched to `(0,height)`. This reverses the image left-to-right. Here is the complete class:

```
import java.awt.*;
import javax.swing.*;
import java.awt.image.BufferedImage;

/**
 * A component for displaying a mirror-reversed line of text.
 * The text will be centered in the available space.  This component
 * is defined as a subclass of JPanel.  It respects any background
 * color, foreground color, and font that are set for the JPanel.
 * The setText(String) method can be used to change the displayed
 * text.  Changing the text will also call revalidate() on this
 * component.
 */
public class MirrorText extends JPanel {

   private String text; // The text displayed by this component.

   private BufferedImage OSC; // Holds an un-reversed picture of the text.

   /**
    * Construct a MirrorText component that will display the specified
    * text in mirror-reversed form.
    */
   public MirrorText(String text) {
      if (text == null)
         text = "";
      this.text = text;
   }
```

```
/**
 * Change the text that is displayed on the label.
 * @param text the new text to display
 */
public void setText(String text) {
   if (text == null)
      text = "";
   if ( ! text.equals(this.text) ) {
      this.text = text;  // Change the instance variable.
      revalidate();      // Tell container to recompute its layout.
      repaint();         // Make sure component is redrawn.
   }
}

/**
 * Return the text that is displayed on this component.
 * The return value is non-null but can be an empty string.
 */
public String getText() {
   return text;
}

/**
 * The paintComponent method makes a new off-screen canvas, if necessary,
 * writes the text to the off-screen canvas, then copies the canvas onto
 * the screen in mirror-reversed form.
 */
public void paintComponent(Graphics g) {
   int width = getWidth();
   int height = getHeight();
   if (OSC == null || width != OSC.getWidth()
                       || height != OSC.getHeight()) {
      OSC = new BufferedImage(width,height,BufferedImage.TYPE_INT_RGB);
   }
   Graphics OSG = OSC.getGraphics();
   OSG.setColor(getBackground());
   OSG.fillRect(0, 0, width, height);
   OSG.setColor(getForeground());
   OSG.setFont(getFont());
   FontMetrics fm = OSG.getFontMetrics(getFont());
   int x = (width - fm.stringWidth(text)) / 2;
   int y = (height + fm.getAscent() - fm.getDescent()) / 2;
   OSG.drawString(text, x, y);
   OSG.dispose();
   g.drawImage(OSC, width, 0, 0, height, 0, 0, width, height, null);
}

/**
 * Compute a preferred size that includes the size of the text, plus
 * a boundary of 5 pixels on each edge.
 */
public Dimension getPreferredSize() {
   FontMetrics fm = getFontMetrics(getFont());
   return new Dimension(fm.stringWidth(text) + 10,
         fm.getAscent() + fm.getDescent() + 10);
```

```
        }

    }  // end MirrorText
```

This class defines the method "`public Dimension getPreferredSize()`". This method is called by a layout manager when it wants to know how big the component would like to be. Standard components come with a way of computing a preferred size. For a custom component based on a *JPanel*, it's a good idea to provide a custom preferred size. Every component has a method `setPreferredSize()` that can be used to set the preferred size of the component. For our *MirrorText* component, however, the preferred size depends on the font and the text of the component, and these can change from time to time. We need a way to compute a preferred size on demand, based on the current font and text. That's what we do by defining a `getPreferredSize()` method. The system calls this method when it wants to know the preferred size of the component. In response, we can compute the preferred size based on the current font and text.

The *StopWatchLabel* and *MirrorText* classes define components. Components don't stand on their own. You have to add them to a panel or other container. The sample program *CustomComponentTest.java* demonstrates using a *MirrorText* and a *StopWatchLabel* component, which are defined by the source code files *MirrorText.java* and *StopWatchLabel.java*.

In this program, the two custom components and a button are added to a panel that uses a *FlowLayout* as its layout manager, so the components are not arranged very neatly. If you click the button labeled "Change Text in this Program", the text in all the components will be changed. You can also click on the stopwatch label to start and stop the stopwatch. When you do any of these things, you will notice that the components will be rearranged to take the new sizes into account. This is known as "validating" the container. This is done automatically when a standard component changes in some way that requires a change in preferred size or location. This may or may not be the behavior that you want. (Validation doesn't always cause as much disruption as it does in this program. For example, in a *GridLayout*, where all the components are displayed at the same size, it will have no effect at all. I chose a *FlowLayout* for this example to make the effect more obvious.) When the text is changed in a *MirrorText* component, there is no automatic validation of its container. A custom component such as *MirrorText* must call the `revalidate()` method to indicate that the container that contains the component should be validated. In the *MirrorText* class, `revalidate()` is called in the `setText()` method.

13.5 Finishing Touches

In this final section, I will present a program that is more complex and more polished than those we have looked at previously. Most of the examples in this book have been "toy" programs that illustrated one or two points about programming techniques. It's time to put it all together into a full-scale program that uses many of the techniques that we have covered, and a few more besides. After discussing the program and its basic design, I'll use it as an excuse to talk briefly about some of the features of Java that didn't fit into the rest of this book.

The program that we will look at is a Mandelbrot Viewer that lets the user explore the famous Mandelbrot set. I will begin by explaining what that means. If you have downloaded the web version of this book, note that the jar file `MandelbrotViewer.jar` is an executable jar file that you can use to run the program as a stand-alone application. The jar file is in the

directory `c13`, which contains all the files for this chapter. The on-line version of this page has two applet versions of the program. One shows the program running on the web page. The other applet appears on the web page as a button; clicking the button opens the program in a separate window.

13.5.1 The Mandelbrot Set

The Mandelbrot set is a set of points in the xy-plane that is defined by a computational procedure. To use the program, all you really need to know is that the Mandelbrot set can be used to make some pretty pictures, but here are the mathematical details: Consider the point that has real-number coordinates `(a,b)` and apply the following computation:

```
Let x = a
Let y = b
Repeat:
   Let newX = x*x - y*y + a
   Let newY = 2*x*y + b
   Let x = newX
   Let y = newY
```

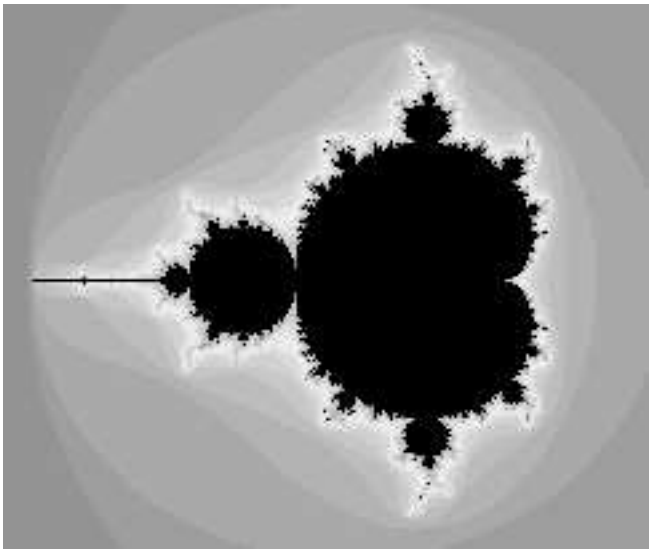
As the loop is repeated, the point `(x,y)` changes. The question is, does `(x,y)` grow without bound or is it trapped forever in a finite region of the plane? If `(x,y)` escapes to infinity (that is, grows without bound), then the starting point `(a,b)` is **not** in the Mandelbrot set. If `(x,y)` is trapped in a finite region, then `(a,b)` is in the Mandelbrot set. Now, it is known that if $x^2 + y^2$ ever becomes strictly greater than 4, then `(x,y)` will escape to infinity. If $x^2 + y^2$ ever becomes bigger than 4 in the above loop, we can end the loop and say that `(a,b)` is definitely not in the Mandelbrot set. For a point `(a,b)` in the Mandelbrot set, the loop will never end. When we do this on a computer, of course, we don't want to have a loop that runs forever, so we put a limit on the number of times that the loop is executed:

```
x = a;
y = b;
count = 0;
while ( x*x + y*y < 4.1 ) {
   count++;
   if (count > maxIterations)
      break;
   double newX = x*x - y*y + a;
   double newY = 2*x*y + b;
   x = newY;
   y = newY;
}
```

After this loop ends, if `count` is less than or equal to `maxIterations`, we can say that `(a,b)` is not in the Mandelbrot set. If `count` is greater than `maxIterations`, then `(a,b)` might or might not be in the Mandelbrot set, but the larger `maxIterations` is, the more likely that `(a,b)` is actually in the set.

To make a picture from this procedure, use a rectangular grid of pixels to represent some rectangle in the plane. Each pixel corresponds to some real number coordinates `(a,b)`. (Use the coordinates of the center of the pixel.) Run the above loop for each pixel. If the `count` goes past `maxIterations`, color the pixel black; this is a point that is possibly in the Mandelbrot set. Otherwise, base the color of the pixel on the value of `count` after the loop ends, using different

colors for different counts. In some sense, the higher the count, the closer the point is to the Mandelbrot set, so the colors give some information about points outside the set and about the shape of the set. However, it's important to understand that the colors are arbitrary and that colored points are **not** in the set. Here is a picture that was produced by the Mandelbrot Viewer program using this computation. The black region is the Mandelbrot set:



When you use the program, you can "zoom in" on small regions of the plane. To do so, just drag the mouse on the picture. This will draw a rectangle around part of the picture. When you release the mouse, the part of the picture inside the rectangle will be zoomed to fill the entire display. If you simply click a point in the picture, you will zoom in on the point where you click by a magnification factor of two. (Shift-click or use the right mouse button to zoom out instead of zooming in.) The interesting points are along the boundary of the Mandelbrot set. In fact, the boundary is infinitely complex. (Note that if you zoom in too far, you will exceed the capabilities of the **double** data type; nothing is done in the program to prevent this.)

Use the "MaxIterations" menu to increase the maximum number of iterations in the loop. Remember that black pixels might or might not be in the set; when you increase "MaxIterations," you might find that a black region becomes filled with color. The "Palette" menu determines the set of colors that are used. Different palettes give very different visualizations of the set. The "PaletteLength" menu determines how many different colors are used. In the default setting, a different color is used for each possible value of `count` in the algorithm. Sometimes, you can get a much better picture by using a different number of colors. If the palette length is less than `maxIterations`, the palette is repeated to cover all the possible values of `count`; if the palette length is greater than `maxIterations`, only part of of the palette will be used. (If the picture is of an almost uniform color, try *decreasing* the palette length, since that makes the color vary more quickly as `count` changes. If you see what look like randomly colored dots instead of bands of color, try *increasing* the palette length.)

If you run the Mandelbrot Viewer program as a stand-alone application, it will have a "File" menu that can be used to save the picture as a PNG image file. You can also save a "param" file which simply saves the settings that produced the current picture. A param file can be read back into the program using the "Open" command.

The Mandelbrot set is named after Benoit Mandelbrot, who was the first person to note the incredible complexity of the set. It is astonishing that such complexity and beauty can arise

out of such a simple algorithm.

13.5.2 Design of the Program

Most classes in Java are defined in packages. While we have used standard packages such as `javax.swing` and `java.io` extensively, almost all of my programming examples have been in the "default package," which means that they are not declared to belong to any named package. However, when doing more serious programming, it is good style to create a package to hold the classes for your program. The Oracle corporation recommends that package names should be based on an Internet domain name of the organization that produces the package. My office computer has domain name `eck.hws.edu`, and no other computer in the world should have the same name. According to Oracle, this allows me to use the package name `edu.hws.eck`, with the elements of the domain name in reverse order. I can also use sub-packages of this package, such as `edu.hws.eck.mdb`, which is the package name that I decided to use for my Mandelbrot Viewer application. No one else—or at least no one else who uses the same naming convention—will ever use the same package name, so this package name uniquely identifies my program.

I briefly discussed using packages in Subsection 2.6.4 and in the context of the programming examples in Section 12.5 Here's what you need to know for the Mandelbrot Viewer program: The program is defined in ten Java source code files. They can be found in the directory *edu/hws/eck/mdb* inside the `source` directory of the web site. (That is, they are in a directory named `mdb`, which is inside a directory named `eck`, which is inside `hws`, which is inside `edu`. The directory structure must follow the package name in this way.) The same directory also contains a file named *strings.properties* that is used by the program and that will be discussed below. For an Integrated Development Environment such as Eclipse, you should just have to add the `edu` directory to your project. To compile the files on the command line, you must be working in the directory that contains the `edu` directory. Use the command

```
javac  edu/hws/eck/mdb/*.java
```

or, if you use Windows,

```
javac  edu\hws\eck\mdb\*.java
```

to compile the source code. The main routine for the stand-alone application version of the program is defined by a class named *Main*. To run this class, use the command:

```
java  edu.hws.eck.mdb.Main
```

This command must also be given in the directory that contains the `edu` directory.

* * *

The work of computing and displaying images of the Mandelbrot set is done in *MandelbrotDisplay.java*. The *MandelbrotDisplay* class is a subclass of *JPanel*. It uses an off-screen canvas to hold a copy of the image. (See Subsection 13.1.1.) The `paintComponent()` method copies this image onto the panel. Then, if the user is drawing a "zoom box" with the mouse, the zoom box is drawn on top of the image. In addition to the image, the class uses a two-dimensional array to store the iteration count for each pixel in the image. If the range of xy-values changes, or if the size of the window changes, all the counts must be recomputed. Since the computation can take quite a while, it would not be acceptable to block the user interface while the computation is being performed. The solution is to do the computation in separate "worker" threads, as discussed in Chapter 12. The program uses one worker thread for each available

processor. When the computation begins, the image is filled with gray. Every so often, about twice a second, the data that has been computed by the computation threads is gathered and applied to the off-screen canvas, and the part of the canvas that has been modified is copied to the screen. A *Timer* is used to control this process—each time the timer fires, the image is updated with any new data that has been computed by the threads. The user can continue to use the menus and even the mouse while the image is being computed.

The file *MandelbrotPanel.java* defines the main panel of the Mandelbrot Viewer window. *MandelbrotPanel* is another subclass of *JPanel*. A *MandelbrotPanel* is mostly filled with a *MandelbrotDisplay*. It also adds a *JLabel* beneath the display. The *JLabel* is used as a "status bar" that shows some information that might be interesting to the user. The *MandelbrotPanel* also defines the program's mouse listener. In addition to handling zooming, the mouse listener puts the x and y coordinates of the current mouse location in the status bar as the user moves or drags the mouse. Also, when the mouse exits the drawing area, the text in the status bar is set to read "Idle". This is the first time that we have seen an actual use for `mouseMoved` and `mouseExited` events. (See Subsection 6.4.2 and Subsection 6.4.4.)

The menu bar for the program is defined in *Menus.java*. Commands in the "File" and "Control" menu are defined as *Actions*. (See Subsection 13.3.1.) Note that among the actions are file manipulation commands that use techniques from Subsection 11.2.3, Subsection 11.5.3, and Subsection 13.1.5. The "MaxIterations," "Palette," and "PaletteLength" menus each contain a group of *JRadioButtonMenuItems*. (See Subsection 13.3.3.) I have tried several approaches for handling such groups, and none of them have satisfied me completely. In this program, I have defined a nested class inside *Menus* to represent each group. For example, the *PaletteManager* class contains the menu items in the "Palette" menu as instance variables. It registers an action listener with each item, and it defines a few utility routines for operating on the menu. The classes for the three menus are very similar and should probably have been defined as subclasses of some more general class.

One interesting point is that the contents of the menu bar are different, depending on whether the program is being run as an applet or as a stand-alone application. Since applets cannot access the file system, there is no "File" menu for an applet. Furthermore, accelerator keys are generally not functional in an applet that is running on a web page, so accelerator keys are only added to menu items if the program is being run in its own window. (See Subsection 13.3.5 for information on accelerators.) To accomplish this, the constructor in the *Menus* class has parameters that tell it whether the menu bar will be used by an applet and whether it will be used in a frame; these parameters are consulted as the menu bar is being built.

A third parameter to the constructor is the *MandelbrotPanel* that is being used in the program. Many of the menu commands operate on this panel or on the *MandelbrotDisplay* that it contains. In order to carry out these commands, the *Menus* object needs a reference to the *MandelbrotPanel*. As for the *MandelbrotDisplay*, the panel has a method `getDisplay()` that returns a reference to the display that it contains. So as long as the menu bar has a reference to the panel, it can obtain a reference to the display. In previous examples, everything was written as one large class file, so all the objects were directly available to all the code. When a program is made up of multiple interacting files, getting access to the necessary objects can be more of a problem.

MandelbrotPanel, *MandelbrotDisplay*, and *Menus* are the main classes that make up the Mandelbrot Viewer program. *MandelbrotFrame.java* defines a simple subclass of *JFrame* that runs the program in its own window. *MandelbrotApplet.java* defines an applet that runs the

program on a web page. (This applet version has an extra "Examples" menu that is discussed in the source code file.) There are a few other classes that I will discuss below.

This brief discussion of the design of the Mandelbrot Viewer has shown that it uses a wide variety of techniques that were covered earlier in this book. In the rest of this section, we'll look at a few new features of Java that were used in the program.

13.5.3 Internationalization

Internationalization refers to writing a program that is easy to adapt for running in different parts of the world. Internationalization is often referred to as ***I18n***, where 18 is the number of letters between the "I" and the final "n" in "Internationalization." The process of adapting the program to a particular location is called ***localization***, and the locations are called ***locales***. Locales differ in many ways, including the type of currency used and the format used for numbers and dates, but the most obvious difference is language. Here, I will discuss how to write a program so that it can be easily translated into other languages.

The key idea is that strings that will be presented to the user should not be coded into the program source code. If they were, then a translator would have to search through the entire source code, replacing every string with its translation. Then the program would have to be recompiled. In a properly internationalized program, all the strings are stored together in one or more files that are separate from the source code, where they can easily be found and translated. And since the source code doesn't have to be modified to do the translation, no recompilation is necessary.

To implement this idea, the strings are stored in one or more ***properties files***. A properties file is just a list of key/value pairs. For translation purposes, the values are strings that will be presented to the user; these are the strings that have to be translated. The keys are also strings, but they don't have to be translated because they will never be presented to the user. Since they won't have to be modified, the key strings can be used in the program source code. Each key uniquely identifies one of the value strings. The program can use the key string to look up the corresponding value string from the properties file. The program only needs to know the key string; the user will only see the value string. When the properties file is translated, the user of the program will see different value strings.

The format of a properties file is very simple. The key/value pairs take the form

```
key.string=value string
```

There are no spaces in the key string or before the equals sign. The value string can contain spaces or any other characters. If the line ends with a backslash ("\"), the value string is continued on the next line; in this case, spaces at the beginning of that line are ignored. One unfortunate detail is that a properties file can contain only plain ASCII characters. The ASCII character set only supports the English alphabet. Nevertheless, a value string can include arbitrary UNICODE characters. Non-ASCII characters just have to be specially encoded. The JDK comes with a program, *native2ascii*, that can convert files that use non-ASCII characters into a form that is suitable for use as a properties file.

Suppose that the program wants to present a string to the user (as the name of a menu command, for example). The properties file would contain a key/value pair such as

```
menu.saveimage=Save PNG Image...
```

where "Save PNG Image..." is the string that will appear in the menu. The program would use the key string, "menu.saveimage", to look up the corresponding value string and would then

use the value string as the text of the menu item. In Java, the look up process is supported by the *ResourceBundle* class, which knows how to retrieve and use properties files. Sometimes a string that is presented to the user contains substrings that are not known until the time when the program is running. A typical example is the name of a file. Suppose, for example, that the program wants to tell the user, "Sorry, the file, *filename*, cannot be loaded", where *filename* is the name of a file that was selected by the user at run time. To handle cases like this, value strings in properties files can include placeholders that will be replaced by strings to be determined by the program at run time. The placeholders take the form "`{0}`", "`{1}`", "`{2}`", ... For the file error example, the properties file might contain:

```
error.cantLoad=Sorry, the file, {0}, cannot be loaded
```

The program would fetch the value string for the key `error.cantLoad`. It would then substitute the actual file name for the placeholder, "`{0}`". Note that when the string is translated, the word order might be completely different. By using a placeholder for the file name, you can be sure that the file name will be put in the correct grammatical position for the language that is being used. Placeholder substitution is not handled by the *ResourceBundle* class, but Java has another class, *MessageFormat*, that makes such substitutions easy.

For the Mandelbrot Viewer program, the properties file is *strings.properties*. (Any properties file should have a name that ends in "`.properties`".) Any string that you see when you run the program comes from this file. For handling value string lookup, I wrote *I18n.java*. The *I18n* class has a static method

```
public static tr( String key, Object... args )
```

that handles the whole process. Here, `key` is the key string that will be looked up in `strings.properties`. Additional parameters, if any, will be substituted for placeholders in the value string. (Recall that the formal parameter declaration "`Object...`" means that there can be any number of actual parameters after `key`; see Subsection 7.2.6.) Typical uses would include:

```
String saveImageCommandText = I18n.tr( "menu.saveimage" );

String errMess = I18n.tr( "error.cantLoad" , selectedFile.getName() );
```

You will see function calls like this throughout the Mandelbrot Viewer source code. The *I18n* class is written in a general way so that it can be used in any program. As long as you provide a properties file as a resource, the only things you need to do are change the resource file name in `I18n.java` and put the class in your own package.

It is actually possible to provide several alternative properties files in the same program. For example, you might include French and Japanese versions of the properties file along with an English version. If the English properties file is named `strings.properties`, then the names for the French and Japanese versions should be `strings_fr.properties` and `strings_ja.properties`. Every language has a two-letter code, such as "fr" and "ja", that is used in constructing properties file names for that language. The program asks for the properties file using the simple name "`strings`". If the program is being run on a Java system in which the preferred language is French, the program will try to load "`strings_fr.properties`"; if that fails, it will look for "`strings.properties`". This means that the program will use the French properties files in a French locale; it will use the Japanese properties file in a Japanese locale; and in any other locale it will use the default properties file.

13.5.4 Events, Events, Events

We have worked extensively with mouse events, key events, and action events, but these are only a few of the event types that are used in Java. The Mandelbrot Viewer program makes use of several other types of events. It also serves as an example of the benefits of event-oriented programming.

Let's start from the following fact: The *MandelbrotDisplay* class knows nothing about any of the other classes that make up the program (with the single exception of one call to the internationalization method `I18n.tr`). Yet other classes are aware of things that are going on in the *MandelbrotDisplay* class. For example, when the size of the display is changed, the new size is reported in the status bar that is part of the *MandelbrotPanel* class. In the *Menus* class, certain menus are disabled when the display begins the computation of an image and are re-enabled when the computation completes. The display doesn't call methods in the *MandelbrotPanel* or *Menus* classes, so how do these classes get their information about what is going on in the display? The answer, of course, is events. The *MandelbrotDisplay* object emits events of various types when various things happen. The *MandelbrotPanel* and *MandelbrotDisplay* objects set up listeners that hear those events and respond to them.

The point is that because events are used for communication, the *MandelbrotDisplay* class is not strongly coupled to the other classes. In fact, it can be used in other programs without any modification and without access to the other classes. The alternative to using events would be to have the display object call methods such as `displaySizeChanged()` or `computationStarted()` in the *MandelbrotPanel* and *MandelbrotFrame* objects to tell them what is going on in the display. This would be strong coupling: Any programmer who wanted to use *MandelbrotDisplay* would also have to use the other two classes or would have to modify the display class so that it no longer refers to the other classes. Of course, not everything can be done with events and not all strong coupling is bad: The *MandelbrotPanel* class refers directly to the *MandelbrotDisplay* class and cannot be used without it—but since the whole purpose of a *MandelbrotPanel* is to hold a *MandelbrotDisplay*, the coupling is not a problem.

* * *

The Mandelbrot Viewer program responds to mouse events on the display. These events are generated by the display object, but the display class itself doesn't care about mouse events and doesn't do anything in response to them. Mouse events are handled by a listener in the *MandelbrotPanel*, which responds to them by zooming the display and by showing mouse coordinates in the status bar.

The status bar also shows the new size of the display whenever that size is changed. To handle this, events of type *ComponentEvent* are used. When the size of a component is changed, a *ComponentEvent* is generated. In the Mandelbrot Viewer program, a *ComponentListener* in the *MandelbrotPanel* class listens for size-change events in the display. When one occurs, the listener responds by showing the new size in the status bar; the display knows nothing about the status bar that shows the display's size.

Component events are also used internally in the *MandelbrotDisplay* class in an interesting way. When the user dynamically changes the size of the display, its size can change several times each second. Normally, a change of display size would trigger the creation of a new off-screen canvas and the start of a new asynchronous computation of the image. However, doing this is a big deal, not something I want to do several times in a second. If you try resizing the program's window, you'll notice that the image doesn't change size dynamically as the window size changes. The same image and off-screen canvas are used as long as the size is changing.

Only about one-third of a second after the size has stopped changing will a new, resized image be produced. Here is how this works: The display sets up a *ComponentEvent* to listen for resize events on itself. When a resize occurs, the listener starts a *Timer* that has a delay of 1/3 second. (See Subsection 6.5.1.) While this timer is running, the `paintComponent()` method does not resize the image; instead, it reuses the image that already exists. If the timer fires 1/3 second later, the image will be resized at that time. However, if another resize event occurs while the first timer is running, then the first timer will be stopped before it has a chance to fire, and a new timer will be started with a delay of 1/3 second. The result is that the image does not get resized until 1/3 second after the size of the window stops changing.

The Mandelbrot Viewer program also uses events of type *WindowEvent*, which are generated by a window when it opens or closes (among other things). One example is in the file *LauncherApplet.java*. This file defines an applet that appears as a button on the web page. The button is labeled "Launch Mandelbrot Viewer". When the user clicks the button, a *MandelbrotFrame* is opened on the screen, and the text on the button changes to "Close Mandelbrot Viewer". When the frame closes, the button changes back to "Launch Mandelbrot Viewer", and the button can be used to open another window. The frame can be closed by clicking the button, but it can also be closed using a "Close" command in the frame's menu bar or by clicking the close box in the frame's title bar. The question is, how does the button's text get changed when the frame is closed by one of the latter two methods? One possibility would be to have the frame call a method in the applet to tell the applet that it is closing, but that would tightly couple the frame class to the applet class. In fact, it's done with *WindowEvents*. A *WindowListener* in the applet listens for close events from the frame. In response to a close event, the text of the button is changed. Again, this can happen even though the frame class knows nothing about the applet class. Window events are also used by *Main.java* to trigger an action that has to be taken when the program is ending; this will be discussed below.

Perhaps the most interesting use of events in the Mandelbrot Viewer program is to enable and disable menu commands based on the status of the display. For this, events of type *PropertyChangeEvent* are used. This event class is part of the "bean" framework that was discussed briefly in Subsection 11.5.2, and class *PropertyChangeEvent* and related classes are defined in the package `java.beans`. The idea is that bean objects are defined by their "properties" (which are just aspects of the state of the bean). When a bean property changes, the bean can emit a *PropertyChangeEvent* to notify other objects of the change. Properties for which property change events are emitted are known technically as ***bound properties***. A bound property has a **name** that identifies that particular property among all the properties of the bean. When a property change event is generated, the event object includes the name of the property that has changed, the previous value of the property, and the new value of the property.

The *MandelbrotDisplay* class has a bound property whose name is given by the constant `MandelbrotDisplay.STATUS_PROPERTY`. A display emits a property change event when its status changes. The possible values of the status property are given by other constants, such as `MandelbrotDisplay.STATUS_READY`. The `READY` status indicates that the display is not currently running a computation and is ready to do another one. There are several menu commands that should be enabled only when the status of the display is `READY`. To implement this, the *Menus* class defines a *PropertyChangeListener* to listen for property change events from the display. When this listener hears an event, it responds by enabling or disabling menu commands according to the new value of the status property.

All of Java's GUI components are beans and are capable of emitting property change events. In any subclass of *Component*, this can be done simply by calling the method

```
public void firePropertyChange(String propertyName,
                                          Object oldValue, Object newValue)
```

For example, the *MandelbrotDisplay* class uses the following method for setting its current status:

```
private void setStatus(String status) {
   if (status == this.status) {
           // Note: Event should be fired only if status actually changes.
      return;
   }
   String oldStatus = this.status;
   this.status = status;
   firePropertyChange(STATUS_PROPERTY, oldStatus, status);
}
```

When writing bean classes from scratch, you have to add support for property change events, if you need them. To make this easier, the `java.beans` package provides the *PropertyChangeSupport* class.

13.5.5 Custom Dialogs

Java has several standard dialog boxes that are defined in the classes *JOptionPane*, *JColorChooser*, and *JFileChooser*. These were introduced in Subsection 6.8.2 and Subsection 11.2.3. Dialogs of all these types are used in the Mandelbrot Viewer program. However, sometimes other types of dialog are needed. In such cases, you can build a custom dialog box.

Dialog boxes are defined by subclasses of the class *JDialog*. Like frames, dialog boxes are separate windows on the screen, and the *JDialog* class is very similar to the *JFrame* class. The big difference is that a dialog box has a ***parent***, which is a frame or another dialog box that "owns" the dialog box. If the parent of a dialog box closes, the dialog box closes automatically. Furthermore, the dialog box will probably "float" on top of its parent, even when its parent is the active window.

Dialog boxes can be either ***modal*** or ***modeless***. When a modal dialog is put up on the screen, the rest of the application is blocked until the dialog box is dismissed. This is the most common case, and all the standard dialog boxes are modal. Modeless dialog boxes are more like independent windows, since they can stay on the screen while the user interacts with other windows. There are no modeless dialogs in the Mandelbrot Viewer program.

The Mandelbrot Viewer program uses two custom dialog boxes. They are used to implement the "Set Image Size" and "Set Limits" commands and are defined by the files *SetImageSizeDialog.java* and *SetLimitsDialog.java*. The "set image size" dialog lets the user enter a new width and height for the Mandelbrot image. The "set limits" dialog lets the user input the minimum and maximum values for x and y that are shown in the image. The two dialog classes are very similar. In both classes, several *JTextFields* are used for user input. Two buttons named "OK" and "Cancel" are added to the window, and listeners are set up for these buttons. If the user clicks "OK", the listener checks whether the inputs in the text fields are legal; if not, an error message is displayed to the user and the dialog stays on the screen. If the input is legal when the user clicks "OK", the dialog is disposed. The dialog is also disposed if the user clicks "Cancel" or clicks the dialog box's close box. The net effect is that the dialog box stays on the screen until the user either cancels the dialog or enters legal values for the inputs and clicks "OK". The program can find out which of these occurred by calling a method named `getInput()` in the dialog object after showing the dialog. This method returns `null` if the dialog was canceled; otherwise it returns the user input.

To make my custom dialog boxes easy to use, I added a `static showDialog()` method to each dialog class. When this function is called, it shows the dialog, waits for it to be dismissed, and then returns the value of the `getInput()` method. This makes it possible to use my custom dialog boxes in much the same way as Java's standard dialog boxes are used.

Custom dialog boxes are not difficult to create and to use, if you already know about frames. I will not discuss them further here, but you can look at the source code file *SetImageSizeDialog.java* as a model.

13.5.6 Preferences

Most serious programs allow the user to set ***preferences***. A preference is really just a piece of the program's state that is saved between runs of the program. In order to make preferences persistent from one run of the program to the next, the preferences could simply be saved to a file in the user's home directory. However, there would then be the problem of locating the file. There would be the problem of naming the file in a way that avoids conflicts with file names used by other programs. And there would be the problem of cluttering up the user's home directory with files that the user shouldn't even have to know about.

To deal with these problems, Java has a standard means of handling preferences. It is defined by the package `java.util.prefs`. In general, the only thing that you need from this package is the class named *Preferences*.

In the Mandelbrot Viewer program, the file *Main.java* has an example of using *Preferences*. `Main.java` runs the stand-alone application version of the program, and its use of preferences applies only when the program is run in that way.

In most programs, the user sets preferences in a custom dialog box. However, the Mandelbrot program doesn't have any preferences that are appropriate for that type of treatment. Instead, as an example, I automatically save a few aspects of the program's state as preferences. Every time the program starts up, it reads the preferences, if any are available. Every time the program terminates, it saves the preferences. (Saving the preferences poses an interesting problem because the program ends when the *MandelbrotFrame* window closes, not when the `main()` routine ends. In fact, the `main()` routine ends as soon as the window appears on the screen. So, it won't work to save the preferences at the end of the main program. The solution is to use events: A listener listens for *WindowEvents* from the frame. When a window-closed event is received, indicating that the program is ending, the listener saves the preferences.)

Preferences for Java programs are stored in some platform-dependent form in some platform-dependent location. As a Java programmer, you don't have to worry about it; the Java preferences system knows where to store the data. There is still the problem of identifying the preferences for one program among all the possible Java programs that might be running on a computer. Java solves this problem in the same way that it solves the package naming problem. In fact, by convention, the preferences for a program are identified by the package name of the program, with a slight change in notation. For example, the Mandelbrot Viewer program is defined in the package `edu.hws.eck.mdb`, and its preferences are identified by the string "/edu/hws/eck/mdb". (The periods have been changed to "/", and an extra "/" has been added at the beginning.)

The preferences for a program are stored in something called a "node." The user preferences node for a given program identifier can be accessed as follows:

```
Preferences root = Preferences.userRoot();
Preferences node = root.node(pathName);
```

where `pathname` is the string, such as "/edu/hws/eck/mdb", that identifies the node. The node itself consists of a simple list of key/value pairs, where both the key and the value are strings. You can store any strings you want in preferences nodes—they are really just a way of storing some persistent data between program runs. In general, though, the key string identifies some particular preference item, and the associated value string is the value of that preference. A *Preferences* object, `prefnode`, contains methods `prefnode.get(key)` for retrieving the value string associated with a given key and `prefnode.put(key,value)` for setting the value string for a given key.

In `Main.java`, I use preferences to store the shape and position of the program's window. This makes the size and shape of the window persistent between runs of the program; when you run the program, the window will be right where you left it the last time you ran it. I also store the name of the directory that is currently selected in the file dialog box that is used by the program for the Save and Open commands. This is particularly satisfying, since the default behavior for a file dialog box is to start in the user's home directory, which is hardly ever the place where the user wants to keep a program's files. With the preferences feature, I can switch to the right directory the first time I use the program, and from then on I'll automatically be back in that directory when I use the program again. You can look at the source code in *Main.java* for the details.

* * *

And that's it.... There's a lot more that I could say about Java and about programming in general, but this book is only "An Introduction to Programming Using Java," and it's time for our journey to end. I hope that it has been a pleasant journey for you, and I hope that I have helped you establish a foundation that you can use as a basis for further exploration.

Exercises for Chapter 13

1. The sample program *PaintWithOffScreenCanvas.java* from Section 13.1 is a simple paint program. Make two improvements to this program: First, add a "File" menu that lets the user open an image file and save the current image in either PNG or JPEG format. Second, add a basic one-level "Undo" command that lets the user undo the most recent operation that was applied to the image. (Do not try to make a multilevel Undo, which would allow the user to undo several operations.)

When you read a file into the program, you should copy the image that you read into the program's off-screen canvas. Since the canvas in the program has a fixed size, you should scale the image that you read so that it exactly fills the canvas.

2. For this exercise, you should continue to work on the program from the previous exercise. Add a "StrokeWidth" menu that allows the user to draw lines of varying thicknesses. Make it possible to use different colors for the interior of a filled shape and for the outline of that shape. To do this, change the "Color" menu to "StrokeColor" and add a "FillColor" menu. (My solution adds two new tools, "Stroked Filled Rectangle" and "Stroked Filled Oval", to represent filled shapes that are outlined with the current stroke.) Add support for filling shapes with transparent color. A simple approach to this is to use a *JCheckboxMenuItem* to select either fully opaque or 50% opaque fill. (Don't try to apply transparency to stokes—it's very difficult to make transparency work correctly for the Curve tool, and in any case, shape outlines look better if they are opaque.) Finally, make the menus more user friendly by adding some keyboard accelerators to some commands and by using *JRadioButtonMenuItems* where appropriate, such as in the color and tool menus. This exercise takes quite a bit of work to get it all right, so you should tackle the problem in pieces.

3. The *StopWatchLabel* component from Subsection 13.4.5 displays the text "Timing..." when the stop watch is running. It would be nice if it displayed the elapsed time since the stop watch was started. For that, you need to create a *Timer*. (See Subsection 6.5.1.) Add a *Timer* to the original source code, *StopWatchLabel.java*, to drive the display of the elapsed time in seconds. Create the timer in the `mousePressed()` routine when the stop watch is started. Stop the timer in the `mousePressed()` routine when the stop watch is stopped. The elapsed time won't be very accurate anyway, so just show the integral number of seconds. You only need to set the text a few times per second. For my *Timer* method, I use a delay of 200 milliseconds for the timer.

4. The custom component example *MirrorText.java*, from Subsection 13.4.5, uses an off-screen canvas to show mirror-reversed text in a *JPanel*. An alternative approach would be to draw the text after applying a transform to the graphics context that is used for drawing. (See Subsection 13.2.5.) With this approach, the custom component can be defined as a subclass of *JLabel* in which the `paintComponent()` method is overridden. Write a version of the `MirrorText` component that takes this approach. The solution is very short, but tricky. Note that the scale transform `g2.scale(-1,1)` does a left-right reflection through the left edge of the component.

5. The sample program *PhoneDirectoryFileDemo.java* from Subsection 11.3.2 keeps data for a "phone directory" in a file in the user's home directory. Exercise 11.5 asked you to

revise that program to use an XML format for the data. Both programs have a simple command-line user interface. For this exercise, you should provide a GUI interface for the phone directory data. You can base your program either on the original sample program or on the modified version from the exercise. Use a *JTable* to hold the data. The user should be able to edit all the entries in the table. Also, the user should be able to add and delete rows. Include either buttons or menu commands that can be used to perform these actions. The delete command should delete the selected row, if any. New rows should be added at the end of the table. For this program, you can use a standard *DefaultTableModel*.

Your program should load data from the file when it starts and save data to the file when it ends, just as the two previous programs do. For a GUI program, you can't simply save the data at the end of the `main()` routine, since `main()` terminates as soon as the window shows up on the screen. You want to save the data when the user closes the window and ends the program. There are several approaches. One is to use a *WindowListener* to detect the event that occurs when the window closes. Another is to use a "Quit" command to end the program; when the user quits, you can save the data and close the window (by calling its `dispose()` method), and end the program. If you use the "Quit" command approach, you don't want the user to be able to end the program simply by closing the window. To accomplish this, you should call

```
frame.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
```

where `frame` refers to the *JFrame* that you have created for the program's user interface. When using a *WindowListener*, you want the close box on the window to close the window, not end the program. For this, you need

```
frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
```

When the listener is notified of a window closed event, it can save the data and end the program.

Most of the *JTable* and *DefaultTableModel* methods that you need for this exercise are discussed in Subsection 13.4.3, but there are a few more that you need to know about. To determine which row is selected in a *JTable*, call `table.getSelectedRow()`. This method returns the row number of the selected row, or returns `-1` if no row is selected. To specify which cell is currently being edited, you can use:

```
table.setRowSelectionInterval(rowNum, rowNum);  // Selects row number rowNum.
table.editCellAt( rowNum, colNum ); // Edit cell at position (rowNum,colNum).
phoneTable.getEditorComponent().requestFocus();  // Put input cursor in cell.
```

One particularly troublesome point is that the data that is in the cell that is currently being edited is not in the table model. The value in the edit cell is not put into the table model until after the editing is finished. This means that even though the user sees the data in the cell, it's not really part of the table data yet. If you lose that data, the user would be justified in complaining. To make sure that you get the right data when you save the data at the end of the program, you have to turn off editing before retrieving the data from the model. This can be done with the following method:

```
private void stopEditing() {
   if (table.getCellEditor() != null)
      table.getCellEditor().stopCellEditing();
}
```

This method must also be called before modifying the table by adding or deleting rows; if such modifications are made while editing is in progress, the effect can be very strange.

Quiz on Chapter 13

1. Describe the object that is created by the following statement, and give an example of how it might be used in a program:

```
BufferedImage OSC = new BufferedImage(32,32,BufferedImage.TYPE_INT_RGB);
```

2. Many programs depend on *resource files*. What is meant by a resource in this sense? Give an example.

3. What is the *FontMetrics* class used for?

4. If a *Color*, `c`, is created as `c = new Color(0,0,255,125)`, what is the effect of drawing with this color?

5. What is *antialiasing*?

6. How is the *ButtonGroup* class used?

7. What does the acronym *MVC* stand for, and how does it apply to the *JTable* class?

8. Describe the picture that is produced by the following `paintComponent()` method:

```
public void paintComponent(Graphics g) {
   super.paintComponent(g);
   Graphics2D g2 = (Graphics2D)g;
   g2.translate( getWidth()/2, getHeight()/2 );
   g2.rotate( 30 * Math.PI / 180 );
   g2.fillRect(0,0,100,100);
}
```

9. What is meant by *Internationalization* of a program?

10. Suppose that the class that you are writing has an instance method `doOpen()` (with no parameters) that is meant to be used to open a file selected by the user. Write a code segment that creates an *Action* that represents the action of opening a file. Then show how to create a button and a menu item from that action.

Appendix: Source Files

This appendix contains a list of the examples appearing in the free, on-line textbook *Introduction to Programming Using Java,* Sixth Edition. The source code files can be found in the on-line version of this book, or in the `source` directory of a download of the web site. You should be able to compile the files and use them. Note however that some of these examples depend on other source files, such as *TextIO.java* and *MosaicPanel.java*, that are not built into Java. These are classes that I have written. **Links to all necessary non-standard source code files are provided below.** To compile a program that uses a non-standard class, the source code file or the compiled class file for that class should be in the same location as the program that uses it. To run the program, you only need the compiled class file. Note that almost all of my classes are meant to be placed in the default package, to be used by programs that are also in the default package. To use them in other packages, you will have to add an appropriate "package" declaration to the beginning of the source code.

The solutions to end-of-chapter exercises are **not** listed in this appendix. Each end-of-chapter exercise has its own Web page, which discusses its solution. The source code of a sample solution of each exercise is given in full on the solution page for that exercise. If you want to compile the solution, you should be able to cut-and-paste the solution out of a Web browser window and into a text editing program. (You can't cut-and-paste from the HTML source of the solution page, since it contains extra HTML markup commands that the Java compiler won't understand; the HTML markup does not appear when the page is displayed in a Web browser.)

Note that most of these examples require Java version 5.0 or later. A few of them were written for older versions, but will still work with current versions. When you compile some of these older programs with current versions of Java, you might get warnings about "deprecated" methods. These warnings are **not** errors. When a method is deprecated, it means that it should not be used in new code, but it has not yet been removed from the language. It is possible that deprecated methods might be removed from the language at some future time, but for now you just get a warning about using them.

Part 1: Text-oriented Examples

Many of the sample programs in the text are based on console-style input/output, where the computer and the user type lines of text back and forth to each other. Some of these programs use the standard output object, `System.out`, for output. Many of them use my non-standard class, *TextIO*, for both input and output. For the programs that use *TextIO*, one of the files *TextIO.java* or TextIO.class must be available when you compile the program, and TextIO.class must be available when you run the program. There is also a GUI version of TextIO; you can find information about it at the end of Part 4, below.

The programs listed here are stand-alone applications, not applets, but I have written applets that simulate many of the programs. These "console applets" appear on Web pages in the on-line version of this textbook. They are based on *TextIOApplet.java*, which provides the same methods as *TextIO*, but in an applet instead of in a stand-alone application. It is straightforward to convert a TextIO program to a TextIOApplet applet. See the comments in the *TextIOApplet.java* file for information about how to do it. One example of this can be found in the file *Interest3Console.java*.

- *Interest.java*, from Section 2.2, computes the interest on a specific amount of money over a period of one year.
- *TimedComputation.java*, from Section 2.3, demonstrates a few basic built-in subroutines and functions.
- *EnumDemo.java*, from Section 2.3, a **very** simple first demonstration of enum types.
- *PrintSquare.java*, from Section 2.4, reads an integer typed in by the user and prints the square of that integer. **This program depends on *TextIO.java*. The same is true for almost all of the programs in the rest of this list.**
- *Interest2.java*, from Section 2.4, calculates interest on an investment for one year, based on user input.
- *CreateProfile.java*, from Section 2.4, a simple demo of output to a file, using TextIO.
- *Interest2WithScanner.java*, from Section 2.4, is a version of *Interest2.java* that uses *Scanner* instead of *TextIO* to read input from the user.
- *Interest3.java*, from Section 3.1, the first example that uses control statements.
- *ThreeN1.java*, from Section 3.2, outputs a 3N+1 sequence for a given stating value.
- *ComputeAverage.java*, from Section 3.3, computes the average value of some integers entered by the user.
- *CountDivisors.java*, from Section 3.4, counts the number of divisors of an integer entered by the user.
- *ListLetters.java*, from Section 3.4, lists all the distinct letters in a string entered by the user.
- *LengthConverter.java*, from Section 3.5, converts length measurements input by the user into different units of measure.
- *ReadNumbersFromFile.java*, from Section 3.7, finds the sum and the average of numbers read from a file. Demonstrates `try..catch` statements.
- *GuessingGame.java*, from Section 4.2, lets the user play guessing games where the computer picks a number and the user tries to guess it. A slight variation of this program, which reports the number of games won by the user, is *GuessingGame2.java*.
- *RowsOfChars.java*, from Section 4.3, a rather useless program in which one subroutine calls another.
- *ThreeN2.java*, from Section 4.4, is an improved 3N+1 program that uses subroutines and prints its output in neat columns.
- *HighLow.java*, from Section 5.4, a simple card game. It uses the classes *Card.java* and *Deck.java*, which are given as examples of object-oriented programming. Also available, the card-related classes *Hand.java* and, from Subsection 5.5.1, *BlackjackHand.java*.
- *RollTwoPairs.java*, from Subsection 5.5.2, uses *PairOfDice.java* to simulate rolling two pairs of dice until the same total is rolled on both pairs.

- *BirthdayProblemDemo.java*, from Section 7.2, demonstrates random access to array elements using the "birthday problem" (how many people do you have to choose at random until two are found whose birthdays are on the same day of the year).
- *ReverseInputNumbers.java*, from Section 7.3, is a short program that illustrates the use of a partially full array by reading some numbers from the user and then printing them in reverse order.
- *ReverseWithDynamicArray.java*, from Section 7.3, reads numbers from the user then prints them out in reverse order. It does this using the class *DynamicArrayOfInt.java* as an example of using dynamic arrays.
- *LengthConverter2.java*, from Section 8.2, converts measurements input by the user to inches, feet, yards, and miles. This improvement on *LengthConverter.java* allows inputs combining several measurements, such as "3 feet 7 inches," and it detects illegal inputs.
- *LengthConverter3.java*, from Section 8.3, is a revision of the previous example that uses exceptions to handle errors in the user's input.
- *TowersOfHanoi.java*, from Section 9.2, prints out the steps in a solution to the Towers of Hanoi problem; an example of recursion.
- *StringList.java*, from Section 9.2, implements a linked list of strings. The program *ListDemo.java* tests this class.
- *PostfixEval.java*, from Section 9.3, evaluates postfix expressions using a stack. Depends on the *StackOfDouble* class defined in *StackOfDouble.java*.
- *SortTreeDemo.java*, from Section 9.4, demonstrates a binary sort tree of strings.
- *SimpleParser1.java*, from Section 9.5, evaluates fully parenthesized expressions input by the user.
- *SimpleParser2.java*, from Section 9.5, evaluates ordinary infix expressions input by the user.
- *SimpleParser3.java*, from Section 9.5, reads infix expressions input by the user and constructs expression trees that represent those expressions.
- *WordListWithTreeSet.java*, from Section 10.2, makes an alphabetical list of words from a file. A *TreeSet* is used to eliminate duplicates and sort the words.
- *SimpleInterpreter.java*, from Section 10.4, demonstrates the use of a *HashMap* as a symbol table in a program that interprets simple commands from the user.
- *WordCount.java*, from Section 10.4, counts the number of occurrences of each word in a file. The program uses several features from the Java Collection Framework.
- *ReverseFile.java*, from Section 11.2, shows how to read and write files in a simple command-line application; uses the non-standard class *TextReader.java*. *ReverseFileWithScanner.java* is a version of the program that uses a *Scanner* instead of a *TextReader*.
- *DirectoryList.java*, from Section 11.2, lists the contents of a directory specified by the user; demonstrates the use of the *File* class.
- *CopyFile.java*, from Section 11.3, is a program that makes a copy of a file, using file names that are given as command-line arguments.
- *PhoneDirectoryFileDemo.java*, from Section 11.3, demonstrates the use of a file for storing data between runs of a program.

- *ReadURL.java*, from Section 11.4, reads and displays the contents of a specified URL, if the URL refers to a text file. *ReadURLApplet.java* is an applet that does much the same thing.
- *ShowMyNetwork.java*, mentioned in Section 11.4, is a short program that prints information about each network interface on the computer where it is run, including IP addresses associated with each interface.
- *DateClient.java* and *DateServer.java*, from Section 11.4, are very simple first examples of network client and server programs.
- *CLChatClient.java* and *CLChatServer.java*, from Section 11.4, demonstrate two-way communication over a network by letting users send messages back and forth; however, no threading is used and the messages must strictly alternate.
- *ThreadTest1.java*, from section Section 12.1, runs one or more threads that all perform the same task, to demonstrate that they run simultaneously and finish in an indeterminate order.
- *ThreadTest2.java*, from section Section 12.1, divides up a task (counting primes) among several threads, to demonstrate parallel processing and the use of synchronization.
- *DateServerWithThreads.java* and *DateServerWithThreadPool.java*, from Section 12.4, are modifications of *DateServer.java* (Subsection 11.4.4) that use threads to handle communication with clients. The first program creates a new thread for each connection. The second uses a thread pool, and it uses a blocking queue to send connections from the main program to the pool. The threaded servers will work with original client program, *DateClient.java*.
- *CLMandelbrotMaster.java*, *CLMandelbrotWorker.java*, and *CLMandelbrotTask.java*, from Section 12.4, are a demonstration of distributed computing in which pieces of a large computation are sent over a network to be computed by "worker" programs.

Part 2: Graphical Examples from the Text

The following programs use a graphical user interface. The majority of them can be run both as stand-alone applications and as applets. (If you have downloaded the web site for this book, note that most of the jar files for Chapter 6 and Chapter 13 are executable jar files which can be run as applications.)

- *GUIDemo.java* is a simple demonstration of some basic GUI components from the Swing graphical user interface library. It appears in the text in Section 1.6, but you won't be able to understand it until you learn about GUI programming.
- *StaticRects.java* is a rather useless applet that simply draws a static image. It is the first example of GUI programming, in Section 3.8. This applet depends on *AnimationBase.java*, which is a simple framework for writing animated applets (even though there's actually no animation.)
- *MovingRects.java*, also from Section 3.8, draws an animated version of the image in the preceding example. This applet depends on *AnimationBase.java*, which is a simple framework for writing animated applets.
- *RandomMosaicWalk.java*, a standalone program that displays a window full of colored squares with a moving disturbance, from Section 4.6. This program depends on *MosaicCanvas.java* and *Mosaic.java*.

- *RandomMosaicWalk2.java* is a version of the previous example, modified to use a few named constants. From Section 4.7.
- *ShapeDraw.java*, from Section 5.5, is an applet that lets the user place various shapes on a drawing area; an example of abstract classes, subclasses, and polymorphism.
- *HelloWorldGUI1.java* and *HelloWorldGUI2.java*, from Section 6.1, show the message "Hello World" in a window, the first one by using the built-in *JOptionPane* class and the second by building the interface "by hand." Another variation, *HelloWorldGUI4.java*, from Section 6.4, uses anonymous inner classes where *HelloWorldGUI1.java* uses named nested classes.
- *HelloWorldApplet.java*, from Section 6.2, defines an applet that displays the message "Hello World" or "Goodbye World". The message changes when the user clicks a button.
- *HelloWorldPanel.java*, from Section 6.2, is a panel that displays a message. The panel is used as the content pane both in the applet *HelloWorldApplet2.java* and in the window of the stand-alone application *HelloWorldGUI3.java*. This example illustrates the technique of using the same panel class in both an applet and a stand-alone application, a technique that will be used in many examples in the rest of the book.
- *ColorChooserApplet.java*, used in Section 6.3 to demonstrate RGB and HSB colors. This applet uses techniques that are not covered until later in the text, and it is not presented as a programming example.
- *RandomStringsApplet.java*, from Section 6.3, shows 25 copies of the string "Java!" (or some other string specified in an applet param) in random colors and fonts. The applet uses *RandomStringsPanel.java* for its content pane, and there is a stand-alone application *RandomStringsApp.java* that uses the same panel class.
- *ClickableRandomStringsApp.java*, from Section 6.4, is similar to *RandomStringsApp.java* except that the window is repainted when the user clicks the window. This is a first example of using a mouse listener. The applet version is *ClickableRandomStringsApplet.java*.
- *SimpleStamper.java*, from Section 6.4, lets the user place rectangles and ovals on a drawing area by clicking with the mouse. The applet version is *SimpleStamperApplet.java*. Both versions use *SimpleStamperPanel.java* for their content panes.
- *SimpleTrackMouse.java*, from Section 6.4, shows information about mouse events. The applet version is *SimpleTrackMouseApplet.java*. Both versions use *SimpleTrackMousePanel.java* for their content panes.
- *SimplePaint.java*, from Section 6.4, lets the user draw curves in a drawing area and select the drawing color from a palette. The class *SimplePaint* can be used either as an applet or as a stand-alone application.
- *RandomArtPanel.java*, from Section 6.5, shows a new random "artwork" every four seconds. This is an example of using a *Timer*. Used in an applet version of the program, *RandomArtApplet.java*, and a stand-alone application version, *RandomArt.java*.
- *KeyboardAndFocusDemo.java*, from Section 6.5, shows how to use keyboard and focus events. This class can be run either has an applet or as a stand-alone application.
- *SubKillerPanel.java*, from Section 6.5, lets the user play a simple game. Uses a timer as well as keyboard and focus events. The applet version is *SubKillerApplet.java*, and the stand-alone application version is *SubKiller.java*.
- *SliderDemo.java*, a simple demo from Section 6.6, is an applet that shows three *JSliders*.

- *TextAreaDemo.java*, from Section 6.6, is an applet that demonstrates a *JTextArea* in a *JScrollPane*.
- *BorderDemo.java*, from Section 6.7, a very simple applet that demonstrates six types of border.
- *SliderAndComboBoxDemo.java*, from Section 6.7, shows how to create several components and lay them out in a *GridLayout*. Can be used either as an applet or as a stand-alone application.
- *SimpleCalc.java*, from Section 6.7, lets the user add, subtract, multiply, or divide two numbers input by the user. A demo of text fields, buttons, and layout with nested subpanels. Can be used either as an applet or as a stand-alone application.
- *NullLayoutDemo.java*, from Section 6.7, shows how to lay out the components in a container for which the layout manager has been set to `null`. Can be used either as an applet or as a stand-alone application.
- *HighLowGUIPanel.java*, from Section 6.7, is a subclass of *JPanel* that implements a GUI version of the card game *HighLow.java*, in which the user sees a playing card and guesses whether the next card will be higher or lower in value. This class depends on *Card.java*, *Hand.java*, and *Deck.java*. A standalone application that uses this panel is in *HighLowGUI.java*. An applet that uses it is in *HighLowGUIApplet.java*.
- *MosaicDrawController.java*, from Section 6.8, demonstrates menus and a color chooser dialog. This is used in a program where the user colors the squares of a mosaic by clicking-and-dragging the mouse. It uses *MosaicPanel.java* to define the mosaic panel itself. *MosaicDrawController* is used in the stand-alone application *MosaicDrawFrame.java*, in the applet *MosaicDrawApplet.java*, and in the applet *MosaicDrawLauncherApplet.java*. The latter applet appears as a button on a web page; clicking the button opens a *MosaicDrawFrame*.
- *SimpleDialogDemo.java*, from Section 6.8, is an applet that demonstrates *JColorChooser* and some dialogs from *JOptionPane*.
- *RandomStringsWithArray.java*, from Section 7.2, shows 25 copies of a message in random colors, sizes, and positions. This is an improved version of *RandomStringsPanel.java* that uses an array to keep track of the data, so that the same picture can be redrawn whenever necessary. (Written only as an applet.)
- *SimpleDrawRects.java*, from Section 7.3, lets the user place colored rectangles in a drawing area. Illustrates the use of an *ArrayList*. An applet version is *SimpleDrawRectsApplet.java*. This program uses and depends on *RainbowPalette.java*.
- *SimplePaint2.java*, from Section 7.3, lets the user draw colored curves and stores the data needed for repainting the drawing surface in a list of type ArrayList<CurveData>. *SimplePaint2Applet.java* runs the program as an applet.
- *Checkers.java*, from Section 7.5, lets two users play a game of checkers against each other. Illustrates the use of a two-dimensional array. (This is the longest program in the book so far, at over 750 lines!) *CheckersApplet.java* runs the program as an applet.
- *Blobs.java*, from Section 9.1, recursively counts groups of colored squares in a grid.
- *DepthBreadth.java*, from Section 9.3, demonstrates stacks and queues.
- *TrivialEdit.java*, from Section 11.3, lets the user edit short text files. This program demonstrates reading and writing files and using file dialogs.

- *SimplePaintWithFiles.java*, from Section 11.3, demonstrates saving data from a program to a file in both binary and character form. The program is a simple sketching program based on *SimplePaint2.java*.
- *SimplePaintWithXML.java* and *SimplePaintWithXMLEncoder.java*, from Section 11.5, demonstrate saving data from a program to a file in XML format. The first program uses an invented XML language, while the second uses an *XMLEncoder* for writing files and an *XMLDecoder* for reading files. These programs are modifications of *SimplePaintWithFiles.java*.
- *XMLDemo.java*, from Section 11.5, is a simple program that demonstrates basic parsing of an XML document and traversal of the Document Object Model representation of the document. The user enters the XML to be parsed in a text area. The trivial class *XMLDemoApplet.java* runs the program as an applet.
- *RandomArtWithThreads.java*, from Section 12.2, uses a thread to drive a very simple animation. Compare to *RandomArtPanel.java*, which does the same thing with a timer.
- *QuicksortThreadDemo.java*, from Section 12.2, demonstrates using a separate thread to perform a computation, with simple inter-thread communication. The program is a visual demo of the QuickSort algorithm. (*QuicksortThreadDemoApplet.java* is a trivial class that shows the program as an applet.)
- *BackgroundComputationDemo.java*, from Section 12.2, demonstrates using a thread running at a lower priority to perform a lengthy computation in the background. (The program computes a visualization of a small piece of the Mandelbrot set, but the particular computation that is done is not important.)
- *MultiprocessingDemo1.java*, from Section 12.2, is a modification of the previous example that uses several threads to perform the background computation. This speeds up the computation on multi-processor machines. (*MultiprocessingDemo1Applet.java* is a trivial class that shows the program as an applet.)
- *MultiprocessingDemo2.java*, from Section 12.3, is a modification of the previous example that decomposes its task into a large number of fairly small subtasks, in order to achieve better load balancing. The program uses a thread pool and a queue of tasks. (*MultiprocessingDemo2Applet.java* is a trivial class that shows the program as an applet.)
- *MultiprocessingDemo3.java*, from Section 12.3, is yet another version of the previous examples. This one uses a pool of threads that run forever, taking tasks from a queue and executing them. To make this possible, a blocking queue is used, defined by the standard *LinkedBlockingQueue* class. *MyLinkedBlockingQueue.java* is a simple example of using `wait()` and `notify()` directly that can be used as a replacement for *LinkedBlockingQueue* in `MultiprocessingDemo3`.
- *TowersOfHanoiWithControls.java*, from Section 12.3, shows an animation of the famous Towers Of Hanoi problem. The user can control the animation with Run/Pause, Next, and StartAgain buttons. The program is an example of using `wait()` and `notify()` directly for communication between threads. (*TowersOfHanoiWithControlsApplet.java* is a trivial class that shows the program as an applet.)
- *ChatSimulation.java*, from Section 12.4 (on-line version only), is an applet that simulates a network chat. There is no networking in this applet. The only point is to demonstrate how a thread could be used to process (simulated) incoming messages.

- *GUIChat.java*, from Section 12.4, is a simple GUI program for chatting between two people over a network. It demonstrates using a thread for reading data from a network connection.
- *netgame.common*, from Section 12.5, is a package that defines a framework for networked games. This framework is used in several examples: A chat room, defined in package *netgame.chat*; a tic-tac-toe game, defined in package *netgame.tictactoe*; and a poker game, defined in package *netgame.fivecarddraw*.
- *HighLowWithImages.java*, from Section 13.1, is a variation of *HighLowGUI.java* that takes playing card images from an image file. Requires the image file *cards.png* and depends on *Card.java*, *Deck.java*, and *Hand.java*. *HighLowWithImagesApplet.java* runs this program as an applet.
- *PaintWithOffScreenCanvas.java*, from Section 13.1, is a little paint program that illustrates the use of a *BufferedImage* as an off-screen canvas. This class, like many other examples from Chapter 12, includes a static nested subclass of *JApplet* that can be used to run the program as an applet. This is, admittedly, a somewhat tricky way to do things. See the source code for information about how to use the applet class.
- *SoundAndCursorDemo.java*, from Section 13.1, lets the user play a few sounds and change the cursor by clicking some buttons. This demonstrates using audio resource files and using an image resource to create a custom cursor. Requires the resource files in the directory *snc_resources*.
- *TransparencyDemo.java*, from Section 13.2, demonstrates using the alpha component of colors. It is also an example of using *FontMetrics*.
- *StrokeDemo.java*, from Section 13.2, demonstrates the use of various *BasicStrokes* for drawing lines and rectangles. Also demonstrates antialiasing.
- *PaintDemo.java*, from Section 13.2, demonstrates using a *GradientPaint* and using a *TexturePaint* to fill a polygon. Uses the image resource files *TinySmiley.png* and *QueenOfHearts.png*.
- *RadioButtonDemo.java*, from Section 13.3, does what its name indicates.
- *ToolBarDemo.java*, from Section 13.3, uses a *JToolBar* that holds a group of 3 radio buttons and a push button. All the buttons use custom icons, and the push button is created from an *Action*.
- *SillyStamper.java*, from Section 13.4, demonstrates using a *JList* of *Icons*. The user can "stamp" images of a selected icon onto a drawing area. This program uses the icon images in the directory *stamper_icons* as resources.
- *StatesAndCapitalsTableDemo.java*, from Section 13.4, is a completely trivial demo of a *JTable*.
- *ScatterPlotTableDemo.java*, from Section 13.4, uses a *TableModel* to customize a *JTable*. The table is a list of xy-points that the user can edit. A scatter plot of the points is displayed.
- *SimpleWebBrowser.java* and *SimpleWebBrowserWithThread.java*, from Section 13.4, implement a simple web browser using *JEditorPane* (which is ridiculously easy). The difference between the programs is that the first loads documents synchronously, which can hang the program in an unpleasant way, while the second uses a thread to load documents asynchronously.

- *SimpleRTFEdit.java*, mentioned but just barely discussed in Section 13.4, lets the user edit RTF files, which are text files in a format that include style information such as bold and italics. This is mainly a simple demo of using *Actions* defined by "editor kits."
- *StopWatchLabel.java* and *MirrorText.java*, from Section 13.4, are classes that implement custom components. *CustomComponentTest.java* is a program that tests them.
- The Mandelbrot program from Section 13.5, which computes and displays visualizations of the Mandelbrot set, is defined by several classes in the package `edu.hws.eck.mdb`. The source code files can be found in the directory *edu/hws/eck/mdb*.

Part 3: End-of-Chapter Applets

This section contains the source code for the applets that are used as decorations at the end of each chapter in the on-line version of this textbook. In general, you should not expect to be able to understand these applets at the time they occur in the text. Some of these are older applets that will work with Java 1.1 or even Java 1.0. They are not meant as examples of good programming practice for more recent versions of Java.

- *Moire.java*, an animated design, shown at the end of Section 1.7. You can use applet parameters to control various aspects of this applet's behavior. Also note that you can click on the applet and drag the pattern around by hand. See the source code for details.
- *JavaPops.java*, from Section 2.6 is a simple animation that shows copies of the string "Java!" in various sizes and colors appearing and disappearing. This is an old applet that depends on an old animation framework named *SimpleAnimationApplet.java*
- *RandomBrighten.java*, showing a grid of colored squares that get more and more red as a wandering disturbance visits them, from the end of Section 4.7. Depends on *MosaicCanvas.java*
- *SymmetricBrighten.java*, a subclass of the previous example that makes a symmetric pattern, from the end of Section 5.7. Depends on *MosaicCanvas.java* and *RandomBrighten.java*.
- *TrackLines.java*, an applet with lines that track the mouse, from Section 6.8.
- *KaleidoAnimate.java*, from Section 7.5, shows moving kaleidoscopic images.
- *SimpleCA.java*, a Cellular Automaton applet, from the end of Section 8.5. This applet depends on the file *CACanvas.java* For more information on cellular automata see http://math.hws.edu/xJava/CA/.
- *TowersOfHanoiGUI.java*, from Section 9.5, graphically shows the solution to the Towers of Hanoi problem with 10 disks. A newer version of this program, with buttons that allow some control of the animation is *TowersOfHanoiWithControls.java*, from Section 12.3.
- *LittlePentominosApplet.java*, from Section 10.5, solves pentominos puzzles using a simple recursive backtracking algorithm. This applet depends on *MosaicPanel.java*. For a much more complex Pentominos applet, see http://math.hws.edu/xJava/PentominosSolver/.
- *Maze.java*, an applet that creates a random maze and solves it, from Section 11.5.
- *LittleQuicksortDemo.java*, from the end of Section 12.5, is a simplified version of *QuicksortThreadDemo.java*, which is discussed in Section 12.2. The applet shows a visualization of the QuickSort algorithm where the goal is to sort a list of colored bars into spectral order from red to violet.

- The applet at the end of Section 13.5 is the same Mandelbrot program that is discussed as an example in that section, with source files in the directory *edu/hws/eck/mdb*.

Part 4: Auxiliary Files

This section lists some of the extra source files that are required by various examples in the previous sections. The files listed here are those which are general enough to be potentially useful in other programming projects. Links to these files are also given above, along with the programs that use them.

- *TextIO.java* defines a class containing some static methods for doing input/output. These methods make it easier to use the standard input and output streams, `System.in` and `System.out`. *TextIO* also supports other input sources and output destinations, such as files. Note that this version of *TextIO* requires Java 5.0 (or higher). The *TextIO* class defined by this file is only useful in a command-line environment, and it might be inconvenient to use in integrated development environments such as Eclipse in which standard input does not work particularly well. In that case, you might want to use the following file instead.
- *TextIO.java for GUI* defines an alternative to the preceding file. It defines a version of *TextIO* with the same set of input and output routines as the original version. But instead of using standard I/O, this GUI version opens its own window, and the input/output is done in that window. Please read the comments at the beginning of the file. (For people who have downloaded this book: The file is located in a directory named TextIO-GUI inside the source directory.)
- *TextIOApplet.java* can be used for writing applets that simulate *TextIO* programs. This makes it possible to write applets that use "console-style" input/output. Such applets are created as subclasses of *TextIOApplet*. See the comments in the file for more information about how to convert a TextIO program into a TextIOApplet applet. An example can be found in the file *Interest3Console.java*.
- *AnimationBase.java*, a class that can be used as a framework for writing animated applets. To use the framework, you have to define a subclass of *AnimationBase*. Section 3.8 has an example.
- *Mosaic.java* contains subroutines for opening and controlling a window that contains a grid of colored rectangles. It depends on *MosaicCanvas.java*. This is a toolbox for writing simple stand-alone applications that use a "mosaic window." This is rather old code, but it is used in examples in Section 4.6 and Section 4.7.
- *MosaicPanel.java* is a greatly improved version of *MosaicCanvas.java* that has many options. This class defines a subclass of JPanel that shows little rectangles arranged in rows and columns. It is used in the "mosaic draw" example in Section 6.8.
- *Expr.java* defines a class *Expr* that represent mathematical expressions involving the variable `x`. It is used in some of the exercises in Chapter 8.
- *netgame.common* is a package that defines a framework for networked games, which is discussed in detail in Section 12.5. (If you have downloaded the web site for this book, you can find the files that make up this package in the subdirectory `netgame/common` of the `source` directory.)

- *PokerRank.java* can be used to assign ranks to hands of cards in poker games. The cards are defined in the class *PokerCard.java*. Both of these classes are part of the package *netgame.fivecarddraw*, which is discussed in Subsection 12.5.4. (The file can be found in the subdirectory `netgame/fivecarddraw` of the `source` directory.)
- *I18n.java* is a class that can be used to help in internationalization of a program. See Subsection 13.5.3. (This file in the subdirectory `edu/hws/eck/mdb` of the `source` directory.)

Glossary

abstract class. A class that cannot be used to create objects, and that exists only for the purpose of creating subclasses. Abstract classes in Java are defined using the modifier `abstract`.

abstract data type (ADT). A data type for which the possible values of the type and the permissible operations on those values are specified, without specifying how the values and operations are to be implemented.

access specifier. A modifier used on a method definition or variable specification that determines what classes can use that method or variable. The access specifiers in Java are `public`, `protected`, and `private`. A method or variable that has no access specifier is said to have "package" visibility.

activation record. A data structure that contains all the information necessary to implement a subroutine call, including the values of parameters and local variables of the subroutine and the return address to which the computer will return when the subroutine ends. Activation records are stored on a stack, which makes it possible for several subroutine calls to be active at the same time. This is particularly important for recursion, where several calls to the same subroutine can be active at the same time.

actual parameter. A parameter in a subroutine call statement, whose value will be passed to the subroutine when the call statement is executed. Actual parameters are also called "arguments".

address. Each location in the computer's memory has an address, which is a number that identifies that location. Locations in memory are numbered sequentially. In modern computers, each byte of memory has its own address. Addresses are used when information is being stored into or retrieved from memory.

algorithm. An unambiguous, step-by-step procedure for performing some task, which is guaranteed to terminate after a finite number of steps.

alpha color component. A component of a color that says how transparent or opaque that color is. The higher the alpha component, the more opaque the color.

API. Application Programming Interface. A specification of the interface to a software package or "toolbox." The API says what classes or subroutines are provided in the toolbox and how to use them.

applet. A type of Java program that is meant to run on a Web page in a Web browser, as opposed to a stand-alone application.

animation. An apparently moving picture created by rapidly showing a sequence of still images, called frames, one after the other. In Java, animations are often driven by *Timer* objects; a new frame of the animation is shown each time the timer fires.

antialiasing. Adjusting the color of pixels to reduce the "jagged" effect that can occur when shapes and text are represented by pixels. For antialiased drawing, when the shape covers only part of a pixel, the color of the shape is blended with the previous color of the pixel. The degree of blending depends on how much of the pixel is covered.

array. A list of items, sequentially numbered. Each item in the list can be identified by its index, that is, its sequence number. In Java, all the items in array must have the same type, called the base type of the array. An array is a random access data structure; that is, you can get directly at any item in the array at any time.

array type. A data type whose possible values are arrays. If *Type* is the name of a type, then Type[] is the array type for arrays that have base type *Type*.

assignment statement. A statement in a computer program that retrieves or computes a value and stores that value in a variable. An assignment statement in Java has the form: ⟨*variable-name*⟩ = ⟨*expression*⟩;

asynchronous event. An event that can occur at an unpredictable time, outside the control of a computer program. User input events, such as pressing a button on the mouse, are asynchronous.

ASCII. American Standard Code for Information Interchange. A way of encoding characters using 7 bits for characters. ASCII code only supports 128 characters, with no accented letters, non-English alphabets, special symbols, or ideograms for non-alphabetic languages such as Chinese. Java uses the much larger and more complete Unicode code for characters.

base case. In a recursive algorithm, a simple case that is handled directly rather than by applying the algorithm recursively.

binary number. A number encoded as a sequence of zeros and ones. A binary number is represented in the "base 2" in the same way that ordinary numbers are represented in the "base 10."

binary tree. A linked data structure that is either empty or consists of a root node that contains pointers to two smaller (possibly empty) binary trees. The two smaller binary trees are called the left subtree and the right subtree.

bit. A single-digit binary number, which can be either 0 or 1.

black box. A system or component of a system that can be used without understanding what goes on inside the box. A black box has an interface and an implementation. A black box that is meant to be used as a component in a system is called a module.

block. In Java programming, a sequence of statements enclosed between a pair of braces, { and }. Blocks are used to group several statements into a single statement. A block can also be empty, meaning that it contains no statements at all and consists of just an empty pair of braces.

blocking operation. An operation, such as reading data from a network connection, is said to "block" if it has to wait for some event to occur. A thread that performs a blocking operation can be "blocked" until the required event occurs. A thread cannot execute any instructions while it is blocked. Other threads in the same program, however, can continue to run.

blocking queue. A queue in which the dequeue operation will block if the queue is empty, until an item is added to the queue. If the blocking queue has a limited capacity, the enqueue operation can also block, if the queue is full.

bottom-up design. An approach to software design in which you start by designing basic components of the system, then combine them into more complex components, and so on.

BufferedImage. A class representing "off-screen canvases," that is, images that are stored in the computer's memory and that can be used for drawing images off-screen.

branch. A control structure that allows the computer to choose among two or more different courses of action. Java has two branch statements: *if* statements and *switch* statements.

byte. A unit of memory that consists of eight bits. One byte of memory can hold an eight-bit binary number.

bytecode. "Java bytecode" is the usual name for the machine language of the Java Virtual Machine. Java programs are compiled into Java bytecode, which can then be executed by the JVM.

charset. A particular encoding of character data into binary form. Examples include UTF-8 and ISO-8859-1.

checked exception. An exception in Java that must be handled, either by a `try..catch` statement or by a `throws` clause on the method that can throw he exception. Failure to handle a checked exception in one way or the other is a syntax error.

class. The basic unit of programming in Java. A class is a collection of static and non-static methods and variables. Static members of a class are part of the class itself; non-static, or "instance," members constitute a blueprint for creating objects, which are then said to "belong" to the class.

client/server. A model of network communication in which a "server" waits at a known address on the network for connection requests that are sent to the server by "clients." This is the basic model for communication using the TCP/IP protocol.

command-line interface. A way of interacting with the computer in which the user types in commands to the computer and the computer responds to each command.

comment. In a computer program, text that is ignored by the computer. Comments are for human readers, to help them understand the program.

compiler. A computer program that translates programs written in some computer language (generally a high-level language) into programs written in machine language.

component. General term for a visual element of a GUI, such as a window, button, or menu. A component is represented in Java by an object belonging to a subclass of the class `java.awt.Component`.

constructor. A special kind of subroutine in a class whose purpose is to construct objects belonging to that class. A constructor is called using the `new` operator, and is not considered to be a "method."

container. A component, such as a *JPanel*, that can contain other GUI components. Containers have `add()` methods that can be used to add components.

contract of a method. The semantic component of the method's interface. The contract specifies the responsibilities of the method and of the caller of the method. It says how to use the method correctly and specifies the task that the method will perform when it is used correctly. The contract of a method should be fully specified by its Javadoc comment.

control structure. A program structure such as an *if* statement or a *while* loop that affects the flow of control in a program (that is, the order in which the instructions in the program are executed).

CPU. Central Processing Unit. The CPU is the part of the program that actually performs calculations and carries out programs.

data structure. An organized collection of data, that can be treated as a unit in a program.

deadlock. A situation in which several threads hang indefinitely, for example because each of them is waiting for some resource that is locked by one of the other threads.

default package. The unnamed package. A class that does not declare itself to be in a named package is considered to be in the default package.

definite assignment. Occurs at a particular point in a program if it is definitely true that a given variable must have been assigned a value before that point in the program. It is only legal to use the value of a local variable if that variable has "definitely" been assigned a value before it is used. For this to be true, the compiler must be able to verify that **every** path through the program from the declaration of the variable to its use must pass through a statement that assigns a value to that variable.

deprecated. Considered to be obsolete, but still available for backwards compatibility. A deprecated Java class or method is still part of the Java language, but it is not advisable to use it in new code. Deprecated items might be removed in future versions of Java.

dialog box. A window that is dependent on another window, called its parent. Dialog boxes are usually popped up to get information from the user or to display a message to the user. Dialog boxes in the Swing API are represented by objects of type *JDialog*.

distributed computing. A kind of parallel processing in which several computers, connected by a network, work together to solve a problem.

dummy parameter. Identifier that is used in a subroutine definition to stand for the value of an actual parameter that will be passed to the subroutine when the subroutine is called. Dummy parameters are also called "formal parameters" (or sometimes just "parameters," when the term "argument" is used instead of actual parameter).

enum. Enumerated type. A type that is defined by listing every possible value of that type. An enum type in Java is a class, and the possible values of the type are objects.

event. In GUI programming, something that happens outside the control of the program, such as a mouse click, and that the program must respond to when it occurs.

exception. An error or exceptional condition that is outside the normal flow of control of a program. In Java, an exception can be represented by an object of type *Throwable* that can be caught and handled in a `try..catch` statement.

fetch-and-execute cycle. The process by which the CPU executes machine language programs. It fetches (that is, reads) an instruction from memory and carries out (that is, executes) the instruction, and it repeats this over and over in a continuous cycle.

flag. A boolean value that is set to true to indicate that some condition or event is true. A single bit in a binary number can also be used as a flag.

formal parameter. Another term for "dummy parameter."

frame. One of the images that make up an animation. Also used as another name for activation record.

function. A subroutine that returns a value.

garbage collection. The automatic process of reclaiming memory that is occupied by objects that can no longer be accessed.

generic programming. Writing code that will work with various types of data, rather than with just a single type of data. The Java Collection Framework, and classes that use similar techniques, are examples of generic programming in Java.

getter. An instance method in a class that is used to read the value of some property of that class. Usually the property is just the value of some instance variable. By convention, a getter is named `getXyz()` where `xyz` is the name of the property.

global variable. Another name for member variable, emphasizing the fact that a member variable in a class exists outside the methods of that class.

graphics context. The data and methods necessary for drawing to some particular destination. A graphics context in Java is an object belonging to the *Graphics* class.

GUI. Graphical User Interface. The modern way of interacting with a computer, in which the computer displays interface components such as buttons and menus on a screen and the user interacts with them—for example by clicking on them with a mouse.

hash table. A data structure optimized for efficient search, insertion, and deletion of objects. A hash table consists of an array of locations, and the location in which an object is stored is determined by that object's "hash code," an integer that can be efficiently computed from the contents of the object.

heap. The section of the computer's memory in which objects are stored.

high level language. A programming language, such as Java, that is convenient for human programmers but that has to be translated into machine language before it can be executed.

HSB. A color system in which colors are specified by three numbers (in Java, real numbers in the range 0.0 to 1.0) giving the hue, saturation, and brightness.

IDE. Integrated Development Environment. A programming environment with a graphical user interface that integrates tools for creating, compiling, and executing programs.

identifier. A sequence of characters that can be used as a name in a program. Identifiers are used as names of variables, methods, and classes.

index. The position number of one item in an array.

implementation. The inside of a black box, such as the code that defines a subroutine.

infinite loop. A loop that never ends, because its continuation condition always evaluates to `true`.

inheritence. The fact that one class can extend another. It then inherits the data and behavior of the class that it extends.

instance of a class. An object that belongs to that class (or a subclass of that class). An object belongs to a class in this sense when the class is used as a template for the object when the object is created by a constructor defined in that class.

instance method. A non-static method in a class and hence a method in any object that is an instance of that class.

instance variable. A non-static variable in a class and hence a variable in any object that is an instance of that class.

interface. As a general term, how to use a black box such as a subroutine. Knowing the interface tells you nothing about what goes on inside the box. "Interface" is also a

reserved word in Java; in this sense, an `interface` is a type that specifies one or more abstract methods. An object that implements the `interface` must provide definitions for those methods.

interpreter. A computer program that executes program written in some computer language by reading instructions from the program, one-by-one, and carrying each one out (by translating it into equivalent machine language instructions).

I/O. Input/Output, the way a computer program communicates with the rest of the world, such as by displaying data to the user, getting information from the user, reading and writing files, and sending and receiving data over a network.

iterator. An object associated with a collection, such a list or a set, that can be used to traverse that collection. The iterator will visit each member of the collection in turn.

Java Collection Framework (JCF). A set of standard classed that implement generic data structures, including *ArrayList* and *TreeSet*, for example.

JDK. Java Development Kit. Basic software that supports both compiling and running Java programs. A JDK includes a command-line programming environment as well as a JRE. You need a JDK if you want to compile Java source code, as well as executing pre-compiled programs.

JRE. Java Runtime Environment. Basic software that supports running standard Java programs that have already been compiled. A JRE includes a Java Virtual Machine and all the standard Java classes.

just-in-time compiler. A kind of combination interpreter/compiler that compiles parts of a program as it interprets them. This allows subsequent executions of the same parts of the program to be executed more quickly than they were the first time. This can result is greatly increased speed of execution. Modern JVMs use a just-in-time compiler.

JVM. Java Virtual Machine. The imaginary computer whose machine language is Java bytecode. Also used to refer to computer programs that act as interpreters for programs written in bytecode; to run Java programs on your computer, you need a JVM.

layout manager. An object whose function is to lay out the components in a container, that is, to set their sizes and locations. Different types of layout managers implement different policies for laying out components.

linked data structure. A collection of data consisting of a number of objects that are linked together by pointers which are stored in instance variables of the objects. Examples include linked lists and binary trees.

linked list. A linked data structure in which nodes are linked together by pointers into a linear chain.

listener. In GUI programming, an object that can be registered to be notified when events of some given type occur. The object is said to "listen" for the events.

literal. A sequence of characters that is typed in a program to represent a constant value. For example, `'A'` is a literal that represents the constant **char** value, A, when it appears in a Java program.

location (in memory). The computer's memory is made up of a sequence of locations. These locations are sequentially numbered, and the number that identifies a particular location is called the address of that location.

local variable. A variable declared within a method, for use only inside that method. A variable declared inside a block is valid from the point where it is declared until the end of block in which the declaration occurs.

loop. A control structure that allows a sequence of instructions to be executed repeatedly. Java has three kinds of loops: *for* loops, *while* loops, and *do* loops

loop control variable. A variable in a *for* loop whose value is modified as the loop is executed and is checked to determine whether or not to end the loop.

machine language. A programming language consisting of instructions that can be executed directed by a computer. Instructions in machine language are encoded as binary numbers. Each type of computer has its own machine language. Programs written in other languages must be translated into a computer's machine language before they can be executed by that computer.

main memory. Programs and data can be stored in a computer's main memory, where they are available to the CPU. Other forms of memory, such as a disk drive, also store information, but only main memory is directly accessible to the CPU. Programs and data from a disk drive have to be copied into main memory before they can be used by the CPU.

map. An associative array; a data structure that associates an object from some collection to each object in some set. In Java, maps are represented by the generic interface *Map<T,S>*

member variable. A variable defined in a class but not inside a method, as opposed to a local variable, which is defined inside some method.

memory. Memory in a computer is used to hold programs and data.

method. Another term for *subroutine*, used in the context of object-oriented programming. A method is a subroutine that is contained in a class or in an object.

module. A component of a larger system that interacts with the rest of the system in a simple, well-defined, straightforward manner.

multitasking. Performing multiple tasks at once, either by switching rapidly back and forth from one task to another or by literally working on multiple tasks at the same time.

multiprocessing. Multitasking in which more than one processor is used, so that multiple tasks can literally be worked on at the same time.

mutual exclusion. Prevents two threads from accessing the same resource at the same time. In Java, this only applies to threads that access the resource in `synchronized` methods or `synchronized` statements. Mutual exclusion can prevent race conditions but introduces the possibility of deadlock.

MVC pattern. The Model/View/Controller pattern, a strategy for dividing responsibility in a GUI component. The model is the data for the component. The view is the visual presentation of the component on the screen. The controller is responsible for reacting to events by changing the model. According to the MVC pattern, these responsibilities should be handled by different objects.

NaN. Not a Number. `Double.NaN` is a special value of type **double** that represents an undefined or illegal value.

node. Common term for one of the objects in a linked data structure.

null. A special pointer value that means "not pointing to anything."

numerical analysis. The field that studies algorithms that use approximations, such as real numbers, and the errors that can result from such approximation.

off-by-one error. A common type of error in which one too few or one too many items are processed, often because counting is not being handled correctly or because the processing stops too soon or continues too long for some other reason.

object. An entity in a computer program that can have data (variables) and behaviors (methods). An object in Java must be created using some class as a template. The class of an object determines what variables and methods it contains.

object type. A type whose values are objects, as opposed to primitive types. Classes and interfaces are object types.

OOP. Object-Oriented Programming. An approach to the design and implementation of computer programs in which classes and objects are created to represent concepts and entities and their interactions.

operating system. The basic software that is always running on a computer, without which it would not be able to function. Examples include Linux, Mac OS, and Windows Vista.

operator. A symbol such as "+", "<=", or "++" that represents an operation that can be applied to one or more values in an expression.

overloading (of operators). The fact that the same operator can be used with different types of data. For example, the "+" operator can be applied to both numbers and strings.

overloading (of method names). The fact that several methods that are defined in the same class can have the same name, as long as they have different signatures.

overriding. Redefining in a subclass. When a subclass provides a new definition of a method that is inherited from a superclass, the new definition is said to override the original definition.

package. In Java, a named collection of related classes and sub-packages, such as `java.awt` and `javax.swing`.

parallel processing. When several tasks are being performed simultaneously, either by multiple processors or by one processor that switches back and forth among the tasks.

parameter. Used to provide information to a subroutine when that subroutine is called. Values of "actual parameters" in the subroutine call statement are assigned to the "dummy parameters" in the subroutine definition before the code in the subroutine is executed.

parameterized type. A type such as *ArrayList<String>* that includes one or more type parameters (*String* in the example).

parsing. Determining the syntactical structure of a string in some language. To parse a string is to determine whether the string is legal according to the grammar of the language, and if so, how it can be created using the rules of the grammar.

partially full array. An array that is used to store varying numbers of items. A partially full array can be represented as a normal array plus a counter to keep track of how many items are actually stored.

pixel. A "picture element" on the screen or in an image. A picture consists of rows and columns of pixels. The color of each pixel can be individually set.

polymorphism. The fact that the meaning of a call to an instance method can depend on the actual type of the object that is used to make the call at run time. That is, if `var` is a variable of object type, then the method that is called by a statement such as

`var.action()` depends on the type of the object to which `var` refers when the statement is executed at run time, not on the type of variable `var`.

pointer. A value that represents an address in the computer's memory, and hence can be thought of as "pointing" to the location that has that address. A variable in Java can never hold an object; it can only hold a pointer to the location where the object is stored. A pointer is also called a "reference."

pragmatics. Rules of thumb that describe what it means to write a *good* program. For example, style rules and guidelines about how to structure a program are part of the pragmatics of a programming language.

precedence. The precedence of operators determines the order in which they are applied, when several operators occur in an expression, in the absence of parentheses.

precondition. A condition that must be true at some point in the execution of a program, in order for the program to proceed correctly from that point. A precondition of a subroutine is something that must be true when the subroutine is called, in order for the subroutine to function properly. Subroutine preconditions are often restrictions on the values of the actual parameters that can be passed into the subroutine.

postcondition. A condition that is known to be true at some point in the execution of a program, as a result of the computation that has come before that point. A postcondition of a subroutine is something that must be true after the subroutine finishes its execution. A postcondition of a function often describe the return value of the function.

primitive type. One of the eight basic built-in data types in Java, **double**, **float**, **long**, **int**, **short**, **byte**, **boolean**, and **char**. A variable of primitive type holds an actual value, as opposed to a pointer to that value.

priority of a thread. An integer associated with a thread that can affect the order in which threads are executed. A thread with greater priority is executed in preference to a thread with lower priority.

producer/consumer. A classic pattern in parallel programming in which one or more producers produce items that are consumed by one or more consumers, and the producers and consumers are meant to run in parallel. The problem is to get items safely and efficiently from the producers to the consumers. In Java, the producer/consumer pattern is implemented by blocking queues.

program. A set of instructions to be carried out by a computer, written in an appropriate programming language. Used as a verb, it means to create such a set of instructions.

programming language. A language that can be used to write programs for a computer. Programming languages range in complexity from machine language to high-level languages such as Java.

protocol. A specification of what constitutes legal communication in a give context. A protocol specifies the format of legal messages, when they can be sent, what kind of reply is expected, and so on.

pseudocode. Informal specification of algorithms, expressed in language that is closer to English than an actual programming language, and usually without filling in every detail of the procedure.

queue. A data structure consisting of a list of items, where items can only be added at one end and removed at the opposite end of the list.

race condition. A source of possible errors in parallel programming, where one thread can cause an error in another thread by changing some aspect of the state of the program that the second thread is depending on (such as the value of variable).

RAM. Random Access Memory. This term is often used as a synonym for the main memory of a computer. Technically, however, it means memory in which all locations are equally accessible at any given time. The term also implies that data can be written to the memory as well as read from it.

recursion. Defining something in terms of itself. In particular, a recursive subroutine is one that calls itself, either directly, or indirectly through a chain of other subroutines. Recursive algorithms work by reducing a complex problem into smaller problems which can be solved either directly or by applying the same algorithm "recursively."

RGB. A color system in which colors are specified by three numbers (in Java, integers in the range 0 to 255) giving the red, green, and blue components of the color.

reference. Another term for "pointer."

return type of a function. The type of value that is returned by that function.

reserved word. A sequence of characters that looks like an identifier but can't be used as an identifier because it has a special meaning in the language. For example, `class`, `public`, and `if` are reserved words in Java.

resource. An image, sound, text, or other data file that is part of a program. Resource files for Java programs are stored on the same class path where the compiled class files for the program are stored.

robust program. A program is robust if it is not only correct, but also is capable of handling errors such as a non-existent file or a failed network connection in a reasonable way.

set. A collection of objects which contains no duplicates. In Java, sets are represented by the generic interface *Set<T>*

scope. The region in a program where the declaration of an identifier is valid.

semantics. Meaning. The semantics rules of a language determine the meaning of strings of symbols (such as sentences or statements) in that language.

sentinel value. A special value that marks the end of a sequence of data values, to indicate the end of the data.

setter. An instance method in a class that is used to set the value of some property of that class. Usually the property is just the value of some instance variable. By convention, a setter is named `setXyz()` where `xyz` is the name of the property.

signature of a method. The name of the method, the number of formal parameters in its definition, and the type of each formal parameter. Method signatures are the information needed by a compiler to tell which method is being called by a given subroutine call statement.

socket. An abstraction representing one end of a connection between two computers on a network. A socket represents a logical connection between computer programs, not a physical connection between computers.

stack. A data structure consisting of a list of items where items can only be added and removed at one end of the list, which is known as the "top" of the stack. Adding an item to a stack is called "pushing," and removing an item is called "popping." The term stack also refers to the stack of activation records that is used to implement subroutine calls.

state machine. A model of computation where an abstract "machine" can be in any of some finite set of different states. The behavior of the machine depends on its state, and the state can change in response to inputs or events. The basic logical structure of a GUI program can often be represented as a state machine.

step-wise refinement. A technique for developing an algorithm by starting with a general outline of the procedure, often expressed in pseudocode, and then gradually filling in the details.

stream. An abstraction representing a source of input data or a destination for output data. Java has four basic stream classes representing input and output of character and binary data. These classes form the foundation for Java's input/output API.

source code. Text written in a high-level programming language, which must be translated into a machine language such as Java bytecode before it can be executed by a computer.

subclass. A class that extends another class, directly or indirectly, and therefore inherits its data and behaviors. The first class is said to be a subclass of the second.

subroutine. A sequence of program instructions that have been grouped together and given a name. The name can then be used to "call" the subroutine. Subroutines are also called *methods* in the context of object-oriented programming.

subroutine call statement. A statement in a program that calls a subroutine. When a subroutine call statement is executed, the computer executes the code that is inside the subroutine.

super. A special variable, automatically defined in any instance method, that refers to the object that contains the method, but considered as belonging to the superclass of the class in which the method definition occurs. `super` gives access to members of the superclass that are hidden by members of the same name in the subclass.

syntax. Grammar. The syntax rules of a language determine what strings of symbols are legal—that is, grammatical—in that language.

TCP/IP. Protocols that are used for network communication on the Internet.

this. A special variable, automatically defined in any instance method, that refers to the object that contains the method.

thread. An abstraction representing a sequence of instructions to be executed one after the other. It is possible for a computer to execute several threads in parallel.

thread pool. A collection of "worker threads" that are available to perform tasks. As tasks become available, they are assigned to threads in the pool. A thread pool is often used with a blocking queue that holds the tasks.

top-down design. An approach to software design in which you start with the problems, as a whole, subdivide it into smaller problems, divide those into even smaller problems, and so on, until you get to problems that can be solved directly.

type. Specifies some specific kind of data values. For example, the type **int** specifies integer data values that can be represented as 32-bit binary numbers. In Java, a type can be a primitive type, a class names, or an interface name. Type names are used to specify the types of variables, of dummy parameters in subroutines, and of return values of subroutines.

type cast. Forces the conversion of a value of one type into another type. For example, in `(int)(6*Math.random())`, the `(int)` is a type-cast operation that converts the **double**

value `(6*Math.random())` into an integer by discarding the fractional part of the real number.

Unicode. A way of encoding characters as binary numbers. The Unicode character set includes characters used in many languages, not just English. Unicode is the character set that is used internally by Java.

URL. Universal Resource Locator; an address for a resource on the Internet, such as a web page.

variable. A named memory location (or sequence of locations) that can be used to store data. A variable is created in a program, and a name is assigned to the variable, in a variable declaration statement. The name can then be used in that program to refer to the memory location, or to the data stored in that memory location, depending on context. In Java, a variable has a *type*, which specifies what kind of data it can hold.

wrapper class. A class such as *Double* or *Integer* that makes it possible to "wrap" a primitive type value in an object belonging to the wrapper class. This allows primitive type values to be used in contexts were objects are required, such as with the Java Collection Framework.

XML. eXtensible Markup Language. A very common and well-supported standard syntax for creating text-based data-representation languages.