Introduction To Programming Using Java

Version 9, JavaFX Edition

David J. Eck

Introduction to Programming Using Java

Version 9.0, JavaFX Edition

May 2022

Author: David J. Eck (eck@hws.edu)

This is an **experimental** ebook version of a free Java Programming textbook. Please excuse any irregularities in formatting. In particular, your reader might have problems with overly wide code segments. Note that PDF and web versions are available on-line at https://math.hws.edu/javanotes/. Source code for examples is **not** included in this ebook, but the ebook does contain links to online copies of the examples. You can email comments to the author.

©1996--2022, David J. Eck.

This work is licensed under a Creative Commons Attribution-Noncommercial-ShareAlike 4.0 License. (This license allows you to redistribute this book in unmodified form for non-commercial purposes. It allows you to make and distribute modified versions for non-commercial purposes, as long as you include an attribution to the original author, clearly describe the modifications that you have made, and distribute the modified work under the same license as the original. Permission might be given by the author for other uses. See the license for full details.) The most recent version of this book is always available, at no charge, for downloading and for on-line use at the Web address: https://math.hws.edu/javanotes/

Introduction to Programming Using Java
Version 9.0, JavaFX Edition, May 2022

Preface

INTRODUCTION TO PROGRAMMING USING JAVA is a free introductory computer programming textbook that uses Java as the language of instruction. This version of the book covers Java 17. 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 more than 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 textbook home page, https://math.hws.edu/javanotes/, has links for downloading both web site and PDF versions of the book. Example programs and solutions to end-of-chapter exercises are part of the web site download and can also be downloaded separately. Readers are encouraged to download the source code for the examples and to read and run the programs as they read the book. Readers are also strongly encouraged to read the exercise solutions if they want to get the most out of this 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. 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. And some people believe that object oriented programming should be emphasized from the very beginning. These are **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 introduced in Chapter 3 with a full treatment 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 generic programming and 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 more advanced features.

Java currently has two major approaches to Graphical User Interface programming: JavaFX and Swing. This edition of the textbook uses JavaFX, but there is an alternative edition that uses Swing. The main differences are in Chapters 6 and 13, which are devoted to GUI programming, but GUI programs and GUI-related material in other chapters also use JavaFX exclusively. For the Swing edition, much of the GUI material is taken from Version 7 of this textbook, with some updating and modification.

Swing is a standard part of Java. JavaFX was introduced as a more modern approach to GUI programming, but it must be downloaded and installed separately from Java itself, which makes it more complicated to use. Swing and JavaFX can both be used to write complex, fully functional GUI programs, and either one is a reasonable choice. Version 8 of this textbook used JavaFX. The alternative edition that uses Swing has been added for Version 9.

GUI programming was never included in the textbook as an end in itself, and it would take another textbook to cover the topic in its entirety. I cover GUI because it is a great example of object-oriented programming, it lets me introduce event-driven programs, and it lets students literally see the effect of the code that they write. JavaFX and Swing both offer good support for all of those purposes.

Version 8 of this textbook originally covered Java 8, but minor updates of that version added notes about new features in Java 9 through Java 16. Version 9 of the book covers Java 17. The main change from Version 8 is the addition of the Swing edition. A section on records has been added to Chapter 7. Many examples have been modified to use text blocks and the new `switch` statement syntax. Some references to the general idea of "abstraction" have been added, such as a short subsection on control abstraction in Section 3.1. A short subsection on final classes and methods has been added to Section 5.5. There are also small corrections and modifications throughout.

The majority of this textbook is valid for Java 8, and the book does not cover features added after Java 8 in as much detail. Note that only Java 8, 11, and 17 are "long-term support" releases. When I introduce a feature that requires Java 11 or Java 17, I will make note of that fact. However, I will never refer to any of the non-long-term-support releases.

The first version of the book was written in 1996, and there have been several versions since then. Version 9 will be the last version. All editions are archived (at least until my retirement in December 2022 and hopefully beyond) at the following Web addresses:

- First edition: https://math.hws.edu/eck/cs124/javanotes1/ (Covers Java 1.0.)
- Second edition: https://math.hws.edu/eck/cs124/javanotes2/ (Covers Java 1.1.)
- Third edition: https://math.hws.edu/eck/cs124/javanotes3/ (Covers Java 1.1.)

- Fourth edition: https://math.hws.edu/eck/cs124/javanotes4/ (Covers Java 1.4.)
- Fifth edition: https://math.hws.edu/eck/cs124/javanotes5/ (Covers Java 5.0.)
- Sixth edition: https://math.hws.edu/eck/cs124/javanotes6/ (Covers Java 5.0, with a bit of 6.0.)
- Seventh edition: https://math.hws.edu/eck/cs124/javanotes7/ (Covers Java 7.)
- Eighth edition: https://math.hws.edu/eck/cs124/javanotes8/ (Covers Java 8, plus a bit of 11 and 17.)
- Ninth edition with JavaFX: https://math.hws.edu/eck/cs124/javanotes9/ (Covers Java 17.)
- Ninth edition with Swing: https://math.hws.edu/eck/cs124/javanotes9-swing/ (Covers Java 17.)

This textbook is **free**, but it is not in the public domain. Version 9 is published under the terms of the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/4.0/. For example, you can:

- Post an unmodified copy of the on-line version on your own Web site (including the parts that list the author and state the license under which it is distributed!).
- Give away unmodified copies of this book or sell them at cost of production, as long as they meet the requirements of the license.
- Use the book as a textbook for a course that you are teaching (even though the students are paying to take that course).
- Make modified copies of the complete book or parts of it and post them on the web or otherwise distribute them non-commercially, provided that attribution to the author is given, the modifications are clearly noted, and the modified copies are distributed under the same license as the original. This includes translations to other languages.

For uses of the book in ways not covered by the license, permission of the author is required.

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. The scripts work on Linux and on MacOS. I have made the complete source files available for download here:

https://math.hws.edu/eck/cs124/downloads/javanotes9-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. However, I have had several requests for the sources and have made them available on an "as-is" basis. For more information about the sources and how they are used see the README file from the source download.

The source files for the JavaFX edition and for the Swing edition are actually generated from a common set of source files that would be even harder to use. If you are interested in exploring them, those ultimate source files can be found on Github. You can browse the files online, or you can clone the git repository, using the following URL:

https://github.com/davidjeck/javanotes9

Professor David J. Eck
Department of Mathematics and Computer Science
Hobart and William Smith Colleges
Email: eck@hws.edu
WWW: https://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 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.

Contents of Chapter 1:

- Section 1: The Fetch and Execute Cycle: Machine Language
- Section 2: Asynchronous Events: Polling Loops and Interrupts
- Section 3: The Java Virtual Machine
- Section 4: Fundamental Building Blocks of Programs
- Section 5: Objects and Object-oriented Programming
- Section 6: The Modern User Interface
- Section 7: The Internet and Beyond
- Quiz on This Chapter

Section 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 value 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. (This is all really somewhat more complicated in modern computers. A typical processing chip these days contains several CPU "cores," which allows it to execute several instructions simultaneously. And access to main memory is speeded up by memory "caches," which can be more quickly accessed than main memory and which are meant to hold data and instructions that the CPU is likely to need soon. However, these complications don't change the basic operation.)

A CPU contains an **Arithmetic Logic Unit**, or ALU, which is the part of the processor that carries out operations such as addition and subtraction. It also holds a small number of **registers**, which are small memory units capable of holding a single number. A typical CPU might have 16 or 32 "general purpose" registers, which hold data values that are immediately accessible for processing, and many machine language instructions refer to these registers. For example, there might be an instruction that takes two numbers from two specified registers, adds those numbers (using the ALU), and stores the result back into a register. And there might be instructions for copying a data value from main memory into a register, or from a register into main memory.

The CPU also includes special purpose registers. The most important of these is the **program counter**, or PC. The CPU uses the PC to keep track of where it is in the program it is executing. The PC simply stores the memory 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. Some machine language instructions modify the value that is stored in the PC. This makes it possible for the computer to "jump" from one point in the program to

another point, which is essential for implementing the program features known as loops and branches that are discussed in Section 1.4.

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. Each zero or one is called a **bit**. 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. In modern computers, each memory location holds a **byte**, which is a sequence of eight bits. A machine language instruction or a piece of data generally consists of several bytes, stored in consecutive memory locations. For example, when a CPU reads an instruction from memory, it might actually read four or eight bytes from four or eight memory locations; the memory address of the instruction is the address of the first of those bytes.

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 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. Each instruction makes the CPU perform some very small task, such as adding two numbers or moving data to or from memory. The CPU does all 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:

Section 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** or **solid state drive** 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 or solid state drive is used for permanent storage of larger amounts of information, but programs have to be loaded from there into main memory before they can actually be executed. A hard disk stores data on a spinning magnetic disk, while a solid state drive is a purely electronic device with no moving parts.)
- 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 generally 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 signaled 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 a 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. Most computers can "forcibly" suspend a thread in this way; computers that can do that 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, and even typical smartphones and tablets, 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 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 Web browsers, are dependent upon the operating system. Common desktop operating systems include Linux, various versions of Windows, and MacOS. Operating systems for smartphones and tablets include Android and iOS.

Section 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, Python, 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. (In fact, Java also depends on the particular operating system under which it is running, since it must work with the operating system to perform certain tasks such as accessing the computer's hardware. But let's ignore that complication here.) 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.

(A compiler is like a human translator who translates an entire book from one language to another, producing a new book in the second language. An interpreter is more like a human interpreter who translates a speech at the

United Nations from one language to another at the same time that the speech is being given.)

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, one of the original home computers was the Commodore 64 or "C64". While you might not find an actual C64, you can find programs that run on other computers—or even in a web browser—that "emulate" one. Such an emulator can run C64 programs by acting as an interpreter for the C64 machine language.

The designers of Java chose to use a combination of compiling and interpreting. 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 a C64 emulator simulates a Commodore 64 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 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 several 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 relatively 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, Java was created with the idea that some programs would 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 the native 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 can be compiled into Java bytecode. However, the combination of Java and Java bytecode is platform-independent, secure, and network-compatible while allowing you to program in a modern high-level object-oriented language.

There are even some other programming 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 Scala, 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 specifically to add better support for some of those languages. And this improvement to the JVM has in turn made possible some new features in Java.

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.

Section 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 consecutive 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. As a programmer, you just need to keep in mind that the name refers to a kind of "box" in memory that can hold data, even though you don'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 there are "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 occur in the program, 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.

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 two endpoints, a rectangle by two of its corners, and an oval by two corners of the rectangle that contains it. The rectangles that I am talking about here have sides that are vertical and horizontal, so that they can be specified by just two points; this is the common meaning of "rectangle" in drawing programs.) 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 message). 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.

Section 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 or, on "touchscreens," your fingers. 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 MacOS, 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 that demonstrates a few standard GUI interface components. When the program is run, a window similar to the picture shown here will open on the computer screen. There are four components in the window with which the user can interact: a button, a checkbox, a text field, and a pop-up menu. These components are labeled. There are a few other components in the window. The labels themselves are components (even though you can't interact with them). The right half of the window is a text area component, which can display multiple lines of text. 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, the whole window can itself be considered to be a "component."

(If you would like to run this program, the source code, GUIDemo.java, is available on line. For more information on using this and other examples from this textbook, see Section 2.6.)

In fact, there are three complete sets of GUI components that can be used with Java. One of these, the AWT or **Abstract Windowing Toolkit**, was part of the original version of Java. The second, which is known as **Swing**, builds on the AWT; it was introduced in Java version 1.2, and was the standard GUI toolkit for many years. The third GUI toolkit, JavaFX, briefly became a standard part of Java in Version 8 but is now distributed separately. JavaFX is meant as a more modern way to write GUI applications, but using it is complicated by the fact that it has to be downloaded and installed separately. This textbook covers JavaFX exclusively, but an alternative version of the textbook is available that covers Swing instead. Either version of the textbook can be a reasonable choice.

When a user interacts with GUI components, "events" are generated. For example, clicking a push button generates an event, and pressing a key on the keyboard generates an event. Each time an event is generated, a message is sent to the program telling it that the event has occurred, and the program responds according to its program. In fact, a typical GUI program consists largely of "event handlers" that tell the program how to respond to various types of events. In the above example, the program 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 just a few of the JavaFX 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 shown here are subclasses, directly or indirectly, of a class called *Control*, which represents general properties that are shared by many JavaFX components. In the diagram, two of the direct subclasses of *Control* themselves have subclasses. The classes *TextField* and *TextArea*, which have certain behaviors in common, are grouped together as subclasses of *TextInputControl*. Similarly *Button* and *CheckBox* are subclasses of

ButtonBase, which represents properties common to both buttons and checkboxes. (*ComboBox*, by the way, is the 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, GUIs, 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.

Section 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 that we know as 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, and 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. (Actually, the claim about uniqueness is not quite true, but the basic idea is valid, and the full truth is complicated.) 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 Microsoft Edge, Firefox, Chrome, and Safari. 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. When Java was first introduced, one of its big attractions was the ability to write **applets**. An applet was a small program that is transmitted over the Internet and that runs on a web page. Applets made it possible for a web page to perform complex tasks and have complex interactions with the user. Alas, applets suffered from a variety of problems, and they are no longer used. There are now other options for running programs on Web pages.

But applets were only one aspect of Java's relationship with the Internet. Java can be 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 support for networking is not Java's only advantage. But many good programming languages have been invented only to be soon forgotten. Java

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, data compression, sound processing, and graphics. And programmers have written Java libraries to provide additional capabilities. Complex, high-performance systems can be developed in Java. For example, Hadoop, a system for large scale data processing, is written in Java. Hadoop has been 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. For example, Java can be used to write programs for Android smartphones (though not for the iPhone). (Android uses Google's own version of Java and does not use the same graphical user interface components as standard Java.)

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 still the most widely taught language at Colleges and Universities. It is similar enough to other popular languages, such as C++, JavaScript, and Python, 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

Question 1:

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

Question 2:

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

Question 3:

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

Question 4:

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

Question 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?

Question 6:

What is a *subroutine?*

Question 7:

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

Question 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.)

Question 9:

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

Question 10:

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

See the Answers

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 "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 should take a look at it before trying to write and use your own programs or trying to use the sample programs in this book.

Contents of Chapter 2:

- Section 1: The Basic Java Application
- Section 2: Variables and the Primitive Types
- Section 3: Strings, Classes, Objects, and Subroutines
- Section 4: Text Input and Output
- Section 5: Details of Expressions
- Section 6: Programming Environments
- Programming Exercises
- Quiz on This Chapter

Section 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**. More correctly, the semantics of a programming language is the set of rules that determine the meaning of a program written in that language. 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**. (I will often use the more common term **style**.)

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, ideally, 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 or inconvenient place. (If you use a command-line interface, like that in a 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 begins with `//` and extends to the end of a line. There is a comment of this form on the last line of the above program. The computer ignores the `//` and everything that follows it on the same line. The second type of comment starts with `/*` and ends with `*/`, and it can extend over more than one line. The first three lines of the program are an example of this second type of comment. (A comment that actually begins with `/**`, like this one does, has special meaning; it is a "Javadoc" comment that can be used to produce documentation for the program. See Subsection 4.6.5.)

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 comment) 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. Those statements make up the script that tells the computer exactly what to do when the program is executed. The `main()` routine can call other 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:

```
optional-package-declaration
optional-imports

public class program-name {

    optional-variable-declarations-and-subroutines

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

    optional-variable-declarations-and-subroutines

}
```

The first two lines have to do with using **packages**. A package is a group of classes. You will start learning about packages in Section 2.4, but our first few example programs will not use them.

The **program-name** in the line that begins "public class" 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.

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.

Section 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 them and the rules for using the names to work with them. That is, the programmer must understand the syntax and the semantics of names.

According to the syntax rules of Java, the most basic names are **identifiers**. Identifiers can be used to name classes, variables, and subroutines. An identifier 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 identifiers:

```
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 words are reserved for special uses in Java, and cannot be used as identifiers. These **reserved words** include: `class`, `public`, `static`, `if`, `else`, `while`, and several dozen other words. (Remember that reserved words are **not** identifiers, since they can't be used as names for things.)

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 this standard 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 in addition to simple identifiers, things in Java can have **compound names** which consist of several simple 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".

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 the data 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 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

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 by assigning a value of the wrong type to a variable. 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. Values of type char are closely related to integer values, since a character is actually stored as a 16-bit integer code number. In fact, we will see that chars in Java can actually be used like integers in certain situations.

It is important to remember that a primitive type value is represented using only a certain, finite number of bits. So, an int can't be an arbitrary integer; it can only be an integer in a certain finite range of values. Similarly, float and double variables can only take on certain values. They are not true real numbers in the mathematical sense. For example, the mathematical constant π can only be approximated by a value of type float or double, since it would

require an infinite number of decimal places to represent it exactly. For that matter, many simple numbers such as 1/3 can only be approximated by floats and doubles.

2.2.3 Literals

A data value is stored in the computer as a sequence of bits. In the computer's memory, it doesn't look anything like a value written on this page. You need a way to include constant values in the programs that you write. In a program, you represent constant values as **literals**. A literal is something that you can type in a program to represent a value. It is a kind of name for a constant value.

For example, to type a value of type char in a program, you must surround it with a pair of single quote marks, such as `'A'`, `'*'`, or `'x'`. The character and the quote marks make up a literal of type char. 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. If you want to store the character A in a variable `ch` of type char, you could do so with the assignment statement

```
ch = 'A';
```

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 numeric 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;`" if `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 binary, octal (base-8), and hexadecimal (base-16) literals. I don't want to cover number bases 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 octal literal 045 represents the number 37, not the number 45. Octal numbers are rarely used, but you need to be aware of what happens when you start a number with a zero. 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`. Finally, binary literals start with `0b` or `0B` and contain only the digits 0 and 1; for example: 0b10110.

As a final complication, numeric literals can include the underscore character ("_"), which can be used to separate groups of digits. For example, the integer constant for two billion could be written 2_000_000_000, which is a good deal easier to decipher than 2000000000. There is no rule about how many digits have to be in each group. Underscores can be especially useful in long binary numbers; for example, `0b1010_1100_1011`.

I will note that hexadecimal numbers can also be used 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.

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 constant 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 less than or equal to 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. For example, if `test` is a variable of type boolean, then both of the following assignment statements are legal:

```
test = true;
test = rate > 0.05;
```

2.2.4 Strings and String Literals

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*. (*String* is a type, but not a primitive type; it is in fact the name of a class, and we will return to that aspect of strings in the next section. The fact that *String* is a class explains why the name begins with an upper case "S" rather than lower case.)

A value of type *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. A

string can contain any number of characters, even zero. A string with no characters is called the **empty string** and is represented by the literal `""`, a pair of double quote marks with nothing between them. Remember the difference between single quotes and double quotes! Single quotes are used for char literals and double quotes for *String* literals! There is a big difference between the *String* `"A"` and the char `'A'`.

Within a string literal, 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.

2.2.5 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 way. 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 0.027 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 = 0.027;
       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 as 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 https://math.hws.edu/javanotes/source. They are also included in the downloadable archive of the web site, in a folder named `source`. The source code for the `Interest` program, for example, can be found in the file Interest.java in subfolder named `chapter2` inside the `source` folder.

Section 2.3

Strings, Classes, Objects, 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 *String*: 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.

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. A subroutine is designed to perform some task. To get that task performed in a program, you can "call" the subroutine using a subroutine call statement. 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 either 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 you can use 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.

Let's first consider subroutines that are part of a class. One of the purposes of a class is to group together some variables and subroutines, which 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`", such as the word "static" in `public static void main...`

When a class contains a static variable or subroutine, the name of the class is part of the full name of the variable or subroutine. For example, the standard class named *System* contains a subroutine named `exit`. To use that subroutine in your program, you must refer to it as `System.exit`. This full name consists of the name of the class that contains the subroutine, followed by a period, followed by the name of the subroutine. This subroutine requires an integer as its parameter, so you would actually use it with a subroutine call statement such as

```
System.exit(0);
```

Calling `System.exit` will terminate the program and shut down the Java Virtual Machine. You could use it if you had some reason to terminate the program before the end of the `main` routine. (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, so you could call `System.exit(1)` to indicate that the program is ending because of an error. The parameter is sent back to the operating system; in practice, the value is usually ignored by the operating system.)

System is just one of many standard classes that come with Java. Another useful class is called *Math*. This class gives us an example of a class that contains static variables: It includes the variables `Math.PI` and `Math.E` whose values are the mathematical constants π and e. *Math* also contains a large number of mathematical "functions." Now, any subroutine performs some specific task, but for some subroutines, the 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 that calls the function.

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

function is a static member subroutine in 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 `x` in this formula represents the parameter to the subroutine; it could be a variable named "x", or it could be replaced by any expression that represents a numerical value. For example, `Math.sqrt(2)` computes the square root of 2, and `Math.sqrt(a*a+b*b)` would be legal as long as `a` and `b` are numeric variables.

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, and `Math.floor(-4.2)` is -5.0. The function `Math.round(x)` returns the integer that is closest to `x`, and `Math.ceil(x)` rounds `x` up to an integer. ("Ceil" is short for "ceiling", the opposite of "floor.")
- `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 effectively random enough for most purposes.) We will find a lot of uses for `Math.random` in future examples.

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, 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. For more accurate timing, you can use `System.nanoTime()` instead. `System.nanoTime()` returns the number of nanoseconds since some arbitrary starting time, where

one nanosecond is one-billionth of a second. However, you should not expect the time to be truly accurate to the nanosecond.

Here is a sample program that performs a few mathematical tasks and reports the time that it takes for the program to run.

```
/**
 * This program performs some mathematical computations
and displays the
 * results.  It also displays the value of the constant
Math.PI.  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
nanoseconds.
        long endTime;   // Time when computations are
done, in nanoseconds.
        long compTime;  // Run time in nanoseconds.
        double seconds; // Time difference, in seconds.

        startTime = System.nanoTime();

        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 =
100:");
        System.out.print("      sin(100)*sin(100) +
cos(100)*cos(100) - 1 is: ");
        System.out.println( Math.sin(100)*Math.sin(100)
                + Math.cos(100)*Math.cos(100) - 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() );

        System.out.print("\nThe value of Math.PI is ");
        System.out.println( Math.PI );

        endTime = System.nanoTime();
        compTime = endTime - startTime;
        seconds = compTime / 1000000000.0;

        System.out.print("\nRun time in nanoseconds was:
");
        System.out.println(compTime);
        System.out.println("(This is probably not
perfectly accurate!");
        System.out.print("\nRun time in seconds was:  ");
        System.out.println(seconds);

    } // end main()

} // end class TimedComputation
```

2.3.2 Classes and Objects

Classes can be containers for static variables and subroutines. However classes also have another purpose. They are used to describe objects. In this role, the class is a **type**, in the same way that int and double are types. That is, the class name can be used to declare variables. Such variables can only hold one type of value. The values in this case are **objects**. An object is a collection of variables and subroutines. Every object has an associated class that tells what "type" of object it is. The class of an object specifies what subroutines and variables that object contains. All objects defined by the same class are similar in that they contain similar collections of variables and subroutines. For example, an object might represent a point in the plane, and it might contain variables named `x` and `y` to represent the coordinates of that point. Every point object would have an `x` and a `y`, but different points would have different values for these variables. A class, named *Point* for

example, could exist to define the common structure of all point objects, and all such objects would then be values of type *Point*.

As another example, let's look again at `System.out.println`. *System* is a class, and `out` is a static variable within that class. However, the value of `System.out` is an **object**, and `System.out.println` is actually the full name of a subroutine that is contained in the object `System.out`. You don't need to understand it at this point, but 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 `println` subroutine that can be used to send information to that destination. The object `System.out` is just one possible destination, and `System.out.println` is a 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 output—can all be used in the same way—through a `println` subroutine. The *PrintStream* class expresses the commonalities among all these objects.

The dual role of classes—to contain static variables and subroutines and to describe objects that can also contain variables and subroutines—can be confusing, and in practice most classes are designed to perform primarily or exclusively in only one of the two possible roles. Fortunately, you will not need to worry too much about it until we start working with objects in a more serious way, in Chapter 5.

By the way, 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. The same is not true for other programming languages, and for the time being, 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.

2.3.3 Operations on Strings

String is a class, and 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 variable `str` of type *String*, the value of `str.length()` is an int equal to the number of characters in the string. 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` are variables 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 Nth 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 is greater than or equal to `s1.length()`.
- `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`. Remember that character positions are numbered starting from zero. Note that the character in position `M` is not included. The returned value is called a substring of `s1`. The subroutine `s1.substring(N)`, with just one parameter, 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 char, `ch`, in `s1`. To find the first occurrence of `x` at or after position `N`, you can use `s1.indexOf(x,N)`. To find the last occurrence of `x` in `s1`, use `s1.lastIndexOf(x)`.
- `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. There is also a function `s1.compareToIgnoreCase(s2)`. (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; it compares individual characters using their Unicode code numbers.)

- `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** changed. 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 that string is concatenated with the other 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.

2.3.4 Text Blocks: Multiline Strings

Representing long strings that extend over several lines can be tedious. The end-of-line characters in the string can be represented by an escaped "n" (that is, by `\n`), and the full string could be broken into several parts that are concatenated together using + operators. For example,

```
String poem = "As I was walking down the stair,\n"
    + "   I met a man who wasn't there.\n"
    + "He wasn't there again today.\n"
    + "   I wish, I wish he'd go away!";
```

However, Java 17 defines a new kind of string literal that makes it easier to represent multiline strings. (Recall that a "literal" is something you type in program to represent a constant value.) The new literals are called **text blocks**. A text block starts with a string of three double-quote characters, followed by optional white space and then a new line. The white space and newline are not part of the string constant that is represented by the text block. The text block is terminated by another string of three double-quote

characters. A text block can be used anywhere an ordinary string literal could be used. For example,

```
String poem = """
    As I was walking down the stair,
       I met a man who wasn't there.
    He wasn't there again today.
       I wish, I wish he'd go away!""";
```

This is easier to write and to read than the previous way of writing the value of `poem`.

Remember that you can't have anything except white space (that is, spaces and tabs) on the same line after the three double-quotes that start the text block. Also, note that any extra white space that occurs at the beginning of every line of the text block is removed from the string that is represented by the literal. In the example, the first four spaces are removed from each line of the text block, but the three extra spaces at the beginning of the second and fourth lines are not removed. This means that you can indent the lines of the text block to match the indentation of your program, without including that indentation in the string represented by the text block.

A text block can include escaped characters such as \t or \\, but aside from the backslash character, '\', nothing in the text block has special meaning. For example, something in the text block that looks like a Java comment is not actually a comment; it is just ordinary characters that are part of the string.

TextBlockDemo.java is a short demo program that uses text blocks.

2.3.5 Introduction to Enums

Java comes with eight built-in primitive types and a huge collection 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, to give an example of defining a new type, we will look here

at one particular case: the ability to define **enums** (short for **enumerated types**).

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 }
```

The definition can be in the same class as the program that uses it, but outside the `main()` routine, or it can be in a separate .java file. (In Java 17, it is also possible to put the definition inside `main()`.) The **enum-type-name** in the definition of an enum 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. An enum value is a **constant**; that is, it represents a fixed value that cannot be changed. 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 such as *Season* 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()`.

Using enums can make a program more readable, since you can use meaningful names for the values. And it can prevent certain types of errors, since a compiler can check that the values assigned to an enum variable are in fact legal values for that variable. For now, you should just 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 for use in this program.

    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.");
    }

}
```

(As I mentioned, an enum can actually be defined in a separate file. The sample program SeparateEnumDemo.java is identical to EnumDemo.java, except that the enum types that it uses are defined in files named Month.java and Day.java.)

Section 2.4

Text Input and Output

WE HAVE SEEN THAT IT IS VERY EASY to display text to the user with the functions `System.out.print` and `System.out.println`. But there is more to say on the topic of outputting text. Furthermore, most programs use data that is input to the program at run time rather than built into the program. So you need to know how to do input as well as output. This section explains how to get data from the user, and it covers output in more detail than we have seen so far. It also has a section on using files for input and output.

2.4.1 Basic Output and Formatted Output

The most basic output function is `System.out.print(x)`, where `x` can be a value or expression of any type. If the parameter, `x`, is not already a string, it is converted to a value of type *String*, and the string is then output to the destination called **standard output**. (Generally, this means that the string is displayed to the user; however, in GUI programs, it outputs to a place where a typical user is unlikely to see it. Furthermore, standard output can be "redirected" to write to a different output destination. Nevertheless, for the type of program that we are working with now, the purpose of `System.out` is to display text to the user.)

`System.out.println(x)` outputs the same text as `System.out.print`, but it follows that text by a line feed, which means that any subsequent output will be on the next line. It is possible to use this function with no parameter, `System.out.println()`, which outputs nothing but a line feed. Note that `System.out.println(x)` is equivalent to

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

You might have noticed that `System.out.print` outputs real numbers with as many digits after the decimal point as necessary, so that for example π is output as 3.141592653589793, and numbers that are supposed to represent money might be output as 1050.0 or 43.575. You might prefer to have these numbers output as, for example, 3.14159, 1050.00, and 43.58. Java has a "formatted output" capability that makes it easy to control how real numbers and other values are printed. A lot of formatting options are available. I will cover just a few of the simplest and most commonly used possibilities here.

The function `System.out.printf` can be used to produce formatted output. (The name "printf," which stands for "print formatted," is copied from the C and C++ programming languages, where this type of output originated.) `System.out.printf` takes one 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 );
```

The output format for a value is given by a **format specifier** in the format string. In this example, the format specifier is `%1.2f`. 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." A very large 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" in 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 "20" in `%20s`, can be added to specify the (minimum) number of characters. The "s" stands for "string," and it can be used for values of type *String*. It can also be used for values of other types; in that case the value is converted into a *String* value in the usual way.

The format specifiers for values of type double are more complicated. An "f", as in `%1.2f`, is used to output a number in "floating-point" form, that is with digits after a 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; a "1" in this position 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 exponential form for very small values and very large values and in floating-point form for other 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`. If you want the output to be left-justified instead of right justified, add a minus sign to the beginning of the format specifier: for example, `%-20s`.

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!).

To include an actual percent sign in the output, use the format specifier `%%` in the format string. You can use `%n` to output a line feed. You can also use a backslash, `\`, as usual in strings to output special characters such as tabs and double quote characters.

The format string in a `System.printf` statement can be given by a text block, which makes it easier to output multiple lines of text. For example,

```
System.out.printf("""
        The equivalent of %d miles is:
            %,d yards,
            %,d feet, or
            %,d inches
        """, miles, 1760*miles, 5280*miles,
12*5280*miles);
```

2.4.2 A First Text Input Example

For some unfathomable reason, Java has traditionally made it difficult 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 predefined 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 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 ideal for use here at the beginning of this course. Java 6 introduced the *Console* class for communicating with the user, but *Console* has its own problems. (It is not always available, and it can only read strings, not numbers.) 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. However, we start with my own version of text input.

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 named *TextIO* that defines subroutines for reading values typed by the user. 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 is defined in a "package" named *textio*. This means that when you look for the file `TextIO.java`, you will find it inside a folder named *textio*. Furthermore, it means that a program that uses *TextIO* must "import" it from the *textio* package. This is done with the import directive

```
import textio.TextIO;
```

This directive must come before the "public class" that begins your program. Most of Java's standard classes, including *Scanner*, are defined in packages and are imported into programs in the same way.

Because *TextIO* is not a built-in part of Java, you must make sure that the *TextIO* 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 folder textio to the same folder that contains your main program. This folder contains the file TextIO.java. See Section 2.6 for information about how to use *TextIO*.

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. The user must type a number and press return before the program can continue. The value that the user typed 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 that number. Note the `import` directive on the first line:

```
import textio.TextIO;

/**
 * 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.println();
        System.out.println("The number that you entered
was " + userInput);
        System.out.println("The square of that number is "
+ square);
        System.out.println();

     } // 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. Note that it is good style to output a question or some other prompt to the user before reading input. Otherwise, the user will have no way of knowing exactly what the computer is waiting for, or even that it is waiting for the user to do something.

2.4.3 Basic TextIO Input Functions

TextIO includes a variety of functions for inputting values of various types. Here are the functions 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 `TextIO.getln()` 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.

`TextIO.getln()` does **not** skip blanks or end-of-lines before reading a value. But the input functions `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. When one of these functions skips over an end-of-line, it outputs a '?' to let the user know that more input is expected.

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, even if they had typed more than one value on the previous line. 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.

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! Note that this program uses formatted output to print out monetary values in their correct format.

```
import textio.TextIO;

/**
 * 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.

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

       System.out.print("Enter the annual interest rate
(as a decimal): ");
       rate = TextIO.getlnDouble();
```

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

        System.out.printf("The amount of interest is
$%1.2f%n", interest);
        System.out.printf("The value after one year is
$%1.2f%n", principal);

    } // end of main()

} // end of class Interest2
```

(You might be wondering why there is only one output routine, `System.out.println`, which can output data values of any type, while there is a separate input routine for each data type. For the output function, the computer can tell what type of value is being output by looking at the parameter. However, the input routines don't have parameters, so the different input routines can only be distinguished by having different names.)

2.4.4 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. Similarly, `System.in` has only one possible source for input data.

TextIO has the ability to write data to files and to read data from files. *TextIO* includes output functions `TextIO.put`, `TextIO.putln`, and `TextIO.putf`. Ordinarily, these functions work exactly like `System.out.print`, `System.out.println`, and `System.out.printf` and are interchangeable with them. However, they can also be used to output text to files and to other destinations.

When you write output using `TextIO.put`, `TextIO.putln`, or `TextIO.putf`, the output is sent to the **current output destination.** By default, the current output destination is standard output. However, *TextIO* has 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 will be created if it does not already exist. Note that if a file with the same name already exists, its previous contents will be erased without any warning!

When you call `TextIO.writeFile`, *TextIO* remembers the file and automatically sends any output from `TextIO.put` or other output functions to that file. If you want to go back to writing to standard output, you can call

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

Here is a simple program that asks the user some questions and outputs the user's responses to a file named "profile.txt." As an example, it uses *TextIO* for output to standard output as well as to the file, but `System.out` could also have been used for the output to standard output.

```
import textio.TextIO;

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 file
        TextIO.putln("Name:            " + name);
        TextIO.putln("Email:           " + email);
        TextIO.putln("Favorite Color:  " + favColor);
        TextIO.putf( "Yearly Income:   %,1.2f%n", salary);

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

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

    }

}
```

In many cases, you want to let the user select the file that will be used for output. You could ask the user to type in the file name, but that is error-prone, and users are more familiar with selecting a file from a file dialog box. The statement

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

will open a typical graphical-user-interface file selection dialog where the user can specify the output file. This also has the advantage of alerting the user if they are about to replace an existing file. It is possible for the user to cancel the dialog box without selecting a file.
`TextIO.writeUserSelectedFile` is a function that returns a boolean value.

The return value is `true` if the user selected a file, and is `false` if the user canceled the dialog box. Your program can check the return value if it needs to know whether it is actually going to write to a file or not.

TextIO can also read from files, as an alternative to reading from standard input. You can specify an 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 with a GUI-style dialog box by saying `TextIO.readUserSelectedFile()`. After you have done this, any input will come from the file instead of being typed by the user. You can go back to reading the user's 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 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 the *TextIO* class 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.

2.4.5 Other TextIO Features

The *TextIO* input functions that we have seen so far can only read one value from a line of input. Sometimes, however, you do want to read more than one value from the same line of input. For example, you might want the user to be able to type something like "42 17" to input the two numbers 42 and 17 on the same line. *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 consume 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.

Previously, I said that `TextIO.getln()` reads a full line of input. Given the way that *TextIO* uses the input buffer, that is not quite true. In fact, if `TextIO.getln()` is called when the buffer is not empty, then it will simply return whatever characters remain in the buffer without getting a new line of input from the user. If you ever want to empty the input buffer and simply discard the input, you can call `TextIO.getln()` as a subroutine, without assigning the returned value to a variable:

```
TextIO.getln();
```

Note, by the way, that although most *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 ints and the user types "42,17", 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.

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.

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 make `TextIO.java` available to any program that uses it. Another option for input is the *Scanner* class. One advantage of using *Scanner* 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, and it has some nice features, but using it 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, since *Scanner* is defined in the package *java.util*, you should add the following import directive 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.) These subroutines can read more than one value from a line, so they are more similar to the "get" versions of *TextIO* subroutines rather than the "getln" versions. 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;

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 (as
a decimal): ");
      rate = stdin.nextDouble();

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

      System.out.printf("The amount of interest is
$%1.2f%n", interest);
      System.out.printf("The value after one year is
$%1.2f%n", principal);

   } // end of main()

} // end of class Interest2WithScanner
```

Note the inclusion of the two lines given above to import `Scanner` and create `stdin`. Also note 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.

Section 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, this book has 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`, `'X'`, and `"Hello"`), 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. The standard class *Integer* contains a couple of constants related to the int data type: `Integer.MAX_VALUE` is the largest possible int, 2147483647, and `Integer.MIN_VALUE` is the smallest int, -2147483648. Similarly, the class *Double* contains some constants related to type double. `Double.MAX_VALUE` is the largest value of type double, and `Double.MIN_VALUE` is the smallest **positive** value. It also has constants to

represent infinite values, `Double.POSITIVE_INFINITY` and `Double.NEGATIVE_INFINITY`, and the special value `Double.NaN` to represent an undefined value. For example, the value of `Math.sqrt(-1)` is `Double.NaN`.

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. I will not cover them all here, but most of the important ones are here.

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 ints, you get an int; if you multiply two doubles, 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 number 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 `A` 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 and only if `N % 2` is zero. (If `N` is odd, then `N % 2` is plus or minus `1`, depending on whether `N` is positive or negative.) More generally, you can check whether an integer `N` is evenly divisible by an integer `M` by checking whether `N % M` is zero.

The `%` operator also works with real numbers. In general, `A % B` is what is left over after you remove as many copies of `B` as possible from `A`. For example, `7.52 % 0.5` is 0.02.

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*. When you use `+` to combine a string with a value of

some other type, it is another example of type conversion, since 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. (If `ch` is `'A'` then `ch++` changes the value of `ch` to `'B'`.)

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 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`, because 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.

Note in particular that the statement `x = x++;` **does not change the value of x**! This is because the value that is being assigned to x is the old value of x, the one that it had before the statement was executed. The net result is that x is incremented but then immediately changed back to its previous value! You also need to remember that `x++` is **not** the same as `x + 1`. The expression `x++` changes the value of x; the expression `x + 1` does not.

This can be confusing, and I have seen many bugs in student programs resulting from the confusion. My advice is: Don't be confused. Use `++` and `--` only as stand-alone statements, not as 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. And functions can return boolean values.

By the way, the operators `==` and `!=` can be used to compare boolean values too. 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 compare strings using subroutines such as `equals()` and `compareTo()`, which were described in Subsection 2.3.3.

Another place where `==` and `!=` don't always work as you would expect is with `Double.NaN`, the constant that represents an undefined value of type double. The value of `x == Double.NaN` is defined to be `false` for any `x`, and `x != Double.NaN` is defined to be `true` in all cases. Those values hold

even when `x` is `Double.NaN`! To test whether a real value `x` is the undefined value `Double.NaN`, use the boolean-valued function `Double.isNaN(x)`.

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. The evaluation has been short-circuited and the division by zero is avoided. (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**. The two expressions must be of the same type. 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 Conversion

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.)

When you type-cast a real number to an integer, the fractional part is discarded. For example, `(int)7.9453` is `7`. 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())`. 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`.)

The type char is almost an integer type. You can assign char values to int variables, and you can assign integer constants in the range 0 to 65535 to char variables. You can also use explicit type-casts between char and the numeric types. For example, `(char)97` is `'a'`, `(int)'+'` is `43`, and (char) ('A' + 2) is 'C'.

Type conversion between *String* and other types cannot be done with type-casts. One way to convert a value of any type into a string is to concatenate it with an empty string. For example, `"" + 42` is the string `"42"`. But a better way is to use the function `String.valueOf(x)`, a static member function in the *String* class. `String.valueOf(x)` returns the value of `x`, converted into a string. For example, `String.valueOf(42)` is the string `"42"`, and if `ch` is a char variable, then `String.valueOf(ch)` is a string of length one containing the single character that is the value of `ch`.

It is also possible to convert certain strings into values of other types. For example, the string `"10"` should be convertible into the int value 10, and the

string "17.42e-2" into the double value 0.1742. In Java, these conversions are handled by built-in functions.

The standard class *Integer* contains a static member function for converting from *String* to int. 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 *Double* includes a function Double.parseDouble. If str is a *String*, then the function call Double.parseDouble(str) tries to convert str into a value of type double. An error occurs if str does not represent a legal double value.

Getting back to assignment statements, 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". Many of Java's operators give rise to similar assignment operators. 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;
```

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 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 -, unary +,
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`?)

Section 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 essentially just one command line environment for Java programming, there are several common IDEs, including Eclipse, IntelliJ IDEA, and BlueJ. 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 using the command line, Eclipse, or BlueJ. (Readers are strongly encouraged to read, compile, and run the examples. Source code for sample programs and solutions to end-of-chapter exercises can be downloaded from the book's web page, https://math.hws.edu/javanotes.)

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.

This textbook is meant for use with Java 17 or later, although the large majority of it it is valid for versions as old as Java 8. In this section, I will try to give you enough information to make it possible to install Java and use it with this textbook. Since Java 8, new versions of Java are released much more frequently than in the past, about twice a year, but only some of the releases are "long-term support" (LTS) releases that will continue to receive bug fixes and security updates over an extended period of time. Java 8, Java 11, and Java 17 are long-term support releases. I recommend that you use Java 17, or Java 11 if you already have that installed, and avoid non-

long-term-support releases unless you have specific need for some feature that they offer.

2.6.1 Getting a JDK

The basic development system for Java programming is usually referred to as a **JDK** (Java Development Kit). Note that Java comes in two versions: a Development Kit version (the JDK) and a Runtime Environment version (the JRE). A Runtime Environment can be used to run Java programs, but it does not allow you to compile your own Java programs. A Development Kit includes the Runtime Environment but also lets you compile programs. (It has become harder find a separate JRE download, but you will still see the term used — sometimes to refer, in fact, to a JDK.) A JDK will include the command line environment that you need to work with Java on the command line. If you decide to use an IDE, you might still need to download a JDK first; note, however, that both the Eclipse IDE and BlueJ now include a JDK, so you do not need to download a separate JDK to use them.

Java was developed by Sun Microsystems, Inc., which was acquired by the Oracle corporation. It is possible to download a JDK directly from Oracle's web site, but starting with Java 11, the Oracle JDK is meant mostly for commercial use. For personal and educational use, it is probably preferable to use OpenJDK, which has the same functionality as the version available from Oracle and is distributed under a fully free, open-source license. Although OpenJDK can be downloaded from https://jdk.java.net/, which is also owned by Oracle, I recommend downloading from Adoptium at this address:

https://adoptium.net/

This site is run by the Eclipse Foundation, and it distributes a version of the OpenJDk under the name "Eclipse Temurin." You will want to download Temurin 17 or later. The JDK comes in different versions, depending on the operating system and type of CPU for your computer, but the Adoptium web page should detect your operating system and CPU, so that you just need to click the "Latest release" download button to get the appropriate version. (If

you need to select the version yourself, note that the CPU type is referred to as "Architecture." You are most likely to be using the "x64" architecture, which works for Intel and AMD CPUs, but the newer M1 Macs use "aarch64," which refers to ARM CPUs.)

The Adoptium site provides installers for MacOS and Windows that make it easier to set up Java on those platforms. (The installer for MacOS is a .pkg file, and the installer for Windows is a .msi file.) If you use the Linux operating system, you can probably install an OpenJDK using the usual software manager for your distribution; if not, you can download a version from Adoptium, but it will just be a compressed archive rather than an installer.

If you download a JDK installer for Windows or MacOS from Adoptium, you can just double-click the installer file to start the installation, if it does not start automatically. If you use the default installation, the installer will set up your computer so that you can use the `javac` and `java` commands on the command line.

The GUI programs in this book use a programming library known as **JavaFX**, which must be downloaded separately from the OpenJDK. You will need to download JavaFX even if you use Eclipse for all of your Java work. (BlueJ comes with JavaFX.) Information about downloading and using JavaFX is given at the end of this section.

(An OpenJDK can also be downloaded as a compressed archive, which you can decompress and place anywhere on your computer. However, to use the `javac` and `java` commands, you will either need to put the bin directory from the OpenJDK directory on your PATH environment variable, or use full path names for the `javac` and `java` commands. The Adoptium installers for Windows and for MacOS will take care of this detail for you.)

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. It is useful to know how to use the command line, and it is particularly important for computer science students, but you can skip this subsection if you plan to do all of your work with Java in an IDE.

To use a command line programming environment, you will have to open a window where you can type commands. In Windows, you can open such a command window by running a program named **cmd**. In MacOS, 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 a very old program called **xterm**; but try looking for "Terminal" in your 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** or **working directory**, which contains files that can be used by the commands that you type. (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 MacOS). When the window first opens, the current directory is your **home directory**, where your personal 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, if the current directory is your home directory, then you can change into your Desktop directory by typing the command `cd Desktop` (and then pressing return).

You might want to create a directory (that is, a folder) to hold your Java work. For example, you might create a directory named `javawork` in your home directory. You can do this using your computer's GUI; another way is to use the command line: Open a command window. If you want to put your work directory in a different folder from your home directory, *cd* into the directory where you want to put it. Then enter the command

`mkdir javawork` to make the directory. When you want to work on programming, open a command window and use the *cd* command to change into your Java 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**. The `javac` command is used to compile Java source code, and `java` is used to run Java programs. These commands, and other commands for working with Java, can be found in a directory named *bin* inside the directory that holds the JDK. If you set things up correctly on your computer, it should recognize these commands when you type them on the command line. Try typing the commands `java -version` and `javac -version`. The output from these commands should tell you which version of Java is being used. If you get a message such as "Command not found," then Java is not correctly configured.

Java should already be configured correctly on Linux, if you have installed Java from the Linux software repositories. The same is true on MacOS and Windows, if you have used an installer from Adoptium.

To test the `javac` command, place a copy of HelloWorld.java into your working directory. (If you downloaded the Web site of this book, you can find it in the directory named `chapter2` inside the directory named `source`; you can use your computer's GUI to copy-and-paste this file into your working directory, or you could `cd` into the `chapter2` folder and work there. Alternatively, you can navigate to HelloWorld.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 (while working in the directory that contains the file `HelloWorld.java`):

```
javac  HelloWorld.java
```

This will compile `HelloWorld.java` and will create a bytecode file named `HelloWorld.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. You will then be able to run the program using the `java` command:

```
java  HelloWorld
```

The computer should respond by outputting the message "Hello World!". Note that although the compiled program is stored in a file named `HelloWorld.class`, the `java` command uses the name of the class, *HelloWorld*, not the name of the file. To run the program, you only need .class file, not the .java file.

Many of the sample programs for this book use *TextIO* to read input from the user (see Subsection 2.4.3). Since *TextIO* is not a standard part of Java, you must make it available to any program that uses it. This means that your working directory should contain a folder named `textio`, and inside that folder should be the file TextIO.java. You can copy `TextIO.java` from this book's source directory, or your can download it from the web site, but you should be sure to place it inside a folder named `textio` in the same directory as the program that uses *TextIO*.

Once you have `TextIO.java` you can run a sample program such as Interest2.java to test user input. First, compile the program with the command

```
javac  Interest2.java
```

If successful, this will create the compiled file named `Interest2.class`. But you will also notice that it creates the file `TextIO.class` inside the `textio` folder, if that file does not already exist. More generally, the `javac` command will compile not just the file that you specify but also any additional Java files that are needed. Once you have `Interest2.class`, you can run it using the command

```
java  Interest2
```

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 each line. When the program ends, you will see the command prompt, and you can enter another command. (Note, by the way, that "`java TextIO`" would not

make sense, since `TextIO` does not have a `main()` routine, and so it is not possible to execute it as a program.)

You can follow a similar procedure to run all of the examples in this book that do not use JavaFX. For running JavaFX programs, see Subsection 2.6.8 below.

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 MacOS, you might download the BBEdit application, which can be used for free. One possibility that will work on any platform is to use **jedit**, a programmer's text editor that is itself written in Java and that can be downloaded for free from www.jedit.org. Another popular cross-platform programming editor is Atom, available from atom.io.

To work on your programs, you can 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 into your working directory. 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 one or more 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, or possibly just some "warnings"; warnings do not stop a program from running. 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 Eclipse IDE

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 a number of 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 daunting. 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. I will discuss Eclipse in some detail and a much simpler alternative, BlueJ, more briefly. IDEs have features that are very useful even for a beginning programmer, although a beginner will want to ignore many of their advanced features.

Unless you happen to be using Oracle's JDK for Java 8, 9, or 10, using Eclipse for JavaFX programs will require some extra configuration. Subsection 2.6.8 discusses using JavaFX in Eclipse. This subsection tells you how to use it for programs that use only standard Java classes.

You can download an Eclipse IDE from eclipse.org. When I install Eclipse, I get the "Eclipse IDE for Java Developers" package (**not** the "installer") from this web page:

https://www.eclipse.org/downloads/packages/

For Windows and Linux, the package download is a compressed archive file. You can simply extract the contents of the archive and place the resulting directory wherever you want it on your computer. You will find the Eclipse application in that directory, and you can start Eclipse by double-clicking the application icon. For MacOS, the download is a .dmg file that contains the Eclipse application. You can open the .dmg file and drag the application to any location that you prefer (probably the Applications folder). Note that the MacOS and Linux packages are available in two versions, for x86_64 and AArch64. This refers to the type of CPU that the computer uses. Newer "M1" Macs can use the AArch64 version; the x86_64 version is for older Macs that use Intel CPUs. For Linux, you are most likely to need the x86_64 version, which will work for Intel and AMD CPUs.

Eclipse is a free program. It is itself written in Java. Recent versions of Eclipse include an OpenJDK (although Eclipse calls it a JRE), so you can use it without downloading a separate JDK. The March 2022 version includes a Java 17 SDK.

The first time you start Eclipse, you will be asked to specify a **workspace**, which is the directory where your work will be stored. You can accept the default name, or provide one of your own. You can use multiple workspaces and select the one that you want to use at startup. When a new workspace is first opened, the Eclipse window will be filled by a large "Welcome" screen that includes links to extensive documentation and tutorials. You should 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**. For example, a view can be a text editor, it can be a place where a program can do I/O, or it can contain a list of your projects. If there are several views in one section of the window, then there will be tabs at the top of the section to select the view

that is displayed in that section. This will happen, for example, if you have several editor views open at the same time.

Each view displays a different type of information. The whole set of views in the window 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 experienced, you might want to use the "Debug Perspective," which has features designed to help you find semantic errors in programs. There are small buttons in the Eclipse toolbar that can be used to switch between perspectives.

The Java Perspective includes a large area in the center of the window that contains text editor views. This is where you will create and edit your 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 one or more other views that you might or might not find useful; I usually close them by clicking the small "X" next to the name of each one. Several other views that will certainly be useful appear under different tabs 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. You can also reset the whole window to its default contents by selecting "Reset Perspective" from 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, you will need to fill in a "Project Name," which can be anything you like. There are two other sections of the window that you might need to pay attention to:

If the project will use JavaFX, you must already have configured the workspace for JavaFX, as described at the end of this section. You should make sure that the JRE configuration that you have created for JavaFX is selected in the "JRE" section of the dialog box, as shown here. For a project that does not require JavaFX, you can set the JRE to be an "execution environment," which should be set to JavaSE 17 or later. And for any program from this textbook, you need to uncheck the option labeled "Create module-info.java file" in the "Module" section. This textbook does **not** use modular programs! Note that the workspace will remember these two settings for the next time that you create a new project and that it is harmless to use a JRE with JavaFX support even for a project that does not use JavaFX.

After entering a project name, and changing the options if necessary, click the "Finish" button. The project should appear in the "Package Explorer" view on the left side of the Eclipse window. Click on the small triangle or plus sign 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. The project also contains the "JRE System Library". This is the collection of standard built-in classes that come with Java; if you have configured the project for JavaFX, it will also include the JavaFX classes.

To run any of the sample Java programs from this textbook, you need to copy the source code file into your Eclipse Java project. Assuming that you have downloaded the source code file onto your computer, you can copy-and-paste

it into the Eclipse window. (Right-click the file icon (or control-click on MacOS); select "Copy" from the pop-up menu; then right-click the project's src folder in the Eclipse window, and select "Paste". Be sure to paste it into the src folder, not into the project itself; files outside the `src` folder are **not** treated as Java source code files.) Alternatively, you can try dragging the file icon from a file browser window onto the src folder in the Eclipse window.

To use the *TextIO*-based examples from this textbook, you must add the source code file TextIO.java to your project. This file has to be in a "package" named `textio`. If you already have `TextIO.java` in a folder named "textio," as described above, then you can simply copy-and-paste the `textio` folder into the "src" folder of your project. Alternatively, you can create the `textio` package using the "New/Package" command from the "File" menu. This will make a folder named "textio" in your project, inside the src folder, and you can then copy-and-paste `TextIO.java` into that folder. In any case, package *textio* should appear under "src" in your project, with `TextIO.java` inside it. (You can drag files from one location to another in the Package Explorer view, if you accidently put a file in the wrong location.)

Once a Java program is in the project, you can open it in an editor by double-clicking the file name in the "Package Explorer" view. To run the program, right-click in the editor window, or on the file name in the Package Explorer view (or control-click in MacOS). 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 section of the Eclipse window below the editor section. If the program uses *TextIO* or *Scanner* 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. (For an easier way to run a program, find and click the small "Run" button in Eclipse's tool bar. This will run either the program in the editor window, the program selected in the Package Explorer view, or the program that was run most recently, depending on context.) Note that when you run a program in Eclipse, it is compiled automatically. There is no separate compilation step.

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, inside a folder named `textio`, in any project that requires it.

To create a new Java program in Eclipse, you must create a new Java class. To do that, 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 in the toolbar 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 that you want to create. 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 window also includes an input box labeled "Package" where you can specify the name of a package to contain the class. Most examples in this book use the "default package," but you can create your own programs in any package. To use the default package, the "Package" input box should be empty. Finally, click the "Finish" button to create the class. The class should appear inside the "src" folder, in a folder corresponding to its package. The new file should automatically open in the editing area so that you can start typing 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; many 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—or simply hover your mouse over the actual error—to get a list of possible fixes, then 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. (You can also click an item with the mouse to select it, 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, when you type a period or certain other characters. For example, if you type "`TextIO.`", 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. 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.

(As a side note, it is possible to use the JDK that is included with Eclipse on the command line. That JDK is a directory inside the Eclipse installation, with a long, complex name. The best way to find the name might be to open The "Installed JREs" section of the Eclipse preferences, as described above, select the built-in JRE in the list of "Installed JREs", and click "Edit." The name of the JDK directory will be in the "JRE home" section of the dialog, and you can copy-and-paste it from there. You need to add /bin— or \bin on Windows—to that directory name to get the name of the directory that contains the JDK command line programs such as `javac`. You can add the

full name of that bin directory to your PATH environment variable, or you can use full path names for the javac and java commands.)

2.6.4 BlueJ

As a simpler alternative to Eclipse, I will mention BlueJ, a small IDE that is designed specifically for people who are learning to program. It is much less complex than Eclipse, but it does have some features that make it useful for education. BlueJ can be downloaded from bluej.org. There are installers available for Windows, for MacOS, and for Debian-based Linux (such as Ubuntu Linux and Linux Mint). The installers include an OpenJDK as well as JavaFX (which is required to run BlueJ), so you will not need to do any additional downloading or configuration. As of March, 2022, BlueJ supports Java 11 but not later versions.

In BlueJ, you can begin a project with the "New Project" command in the "Project" menu. A BlueJ project is simply a folder. When you create a project, you will have to select a folder name that does not already exist. The folder will be created and a window will be opened to show the contents of the folder. Files are shown as icons in the BlueJ window. You can drag .java files from the file system onto that window to add files to the project; they will be copied into the project folder as well as shown in the window. You can also copy files directly into the project folder, but BlueJ won't see them until the next time you open the project. When you restart BlueJ, it should show the project that you were working on most recently, but you can open any project with a command from the "Project" menu.

There is a button in the project window for creating a new class. An icon for the class is added to the window, and a .java source code file is created in the project folder. The file is not automatically opened for editing. To edit a file, double-click its icon in the project window. An editor will be opened in a separate window. (A newly created class will contain some default code that you probably don't want; you can erase it and add a `main()` routine instead.) The BlueJ editor does not show errors as you type. Errors will be reported when you compile the program. Also, it does not offer automatic fixes for errors. It has a less capable version of Eclipse's Content Assist,

which seems only to work for getting a list of available subroutines in a class or object; call up the list by hitting Control-Space after typing the period following the name of a class or object.

An editor window contains a button for compiling the program in the window. There is also a compile button in the project window, which compiles all the classes in the project.

To run a program, it must already be compiled. Right-click the icon of a compiled program. In the menu that pops up, you will see "`void main(String[] args)`" or, for a JavaFX program, "Run JavaFX Application"). Select that option from the menu to run the program.

One of the neatest features of BlueJ is that you can actually use it to run any subroutine, not just `main()`. If a class contains other subroutines, you will see them in the list that you get by right-clicking its icon. A pop-up dialog allows you to enter any parameters required by the routine, and if the routine is a function, you will get another dialog box after the routine has been executed to tell you its return value. This allows easy testing of individual subroutines. Furthermore, you can also use BlueJ to create new objects from a class. An icon for the object will be added at the bottom of the project window, and you can right-click that icon to get a list of subroutines in the object. This will, of course, not be useful to you until we get to object-oriented programming in Chapter 5.

2.6.5 The Problem of Packages

Every class in Java is contained in something called a **package**. Classes that are not explicitly put into a package are in the "default" package. All of Java's standard classes are in named packages. This includes even classes like *String* and *System*, which are in a package named *java.lang*. Classes in *java.lang* are automatically imported into any Java file, but classes in other packages must be imported using an `import` directive. My *TextIO* class is in a package named *textio*, and it must be imported into a program that wants to use it. I will discuss packages in greater detail in Section 4.6. For now, you just need to know some basic facts.

Although most of my examples are in the default package, the use of the default package is in fact discouraged, according to official Java style guidelines. Nevertheless, I have chosen to use it, since it seems easier for beginning programmers to avoid packages as much as possible, at least at first. If Eclipse tries to put a class into a package, you can delete the package name from the class-creation dialog to get it to use the default package instead. But 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;
```

For example, the source code for *TextIO* begins with "package textio;". I put *TextIO* in a package because a class that is in a non-default package cannot use a class from the default package. That is, if *TextIO* were in the default package, then it could **only** be used by programs that are also in the default package. (In fact, in earlier versions of this textbook, *TextIO* was in the default package. I moved it to package *textio* as of Version 8 of the book.)

When packages are used in a command-line environment, some complications arise. For example, if a program 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 working in the main directory, not in the subdirectory. When you compile the source code file, you have to include the name of the directory in the command: For example, for a program in package *test.pkg* use "`javac test/pkg/ClassName.java`" on Linux or MacOS, 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.

2.6.6 About jshell

I will mention one more command-line tool for working with Java: *jshell*. The `jshell` command is a standard part of the JDK (after Java 8). If you can

use the `javac` and `java` commands on the command line, then you can also use `jshell`. The purpose of `jshell` is to let you type in and execute Java code without the bother of creating a .java file and writing a main program. To start `jshell`, just enter the command on a line by itself. You will get a jshell prompt where you can enter either a Java statement or a Java expression. If you enter a statement, it will be executed. If you enter an expression, its value will be printed. You do not have to place a semicolon at the end of a line. Here is a short example of a jshell session.

```
$ jshell
|  Welcome to JShell -- Version 17.0.2
|  For an introduction type: /help intro

jshell> System.out.println("Hello World")
Hello World

jshell> int x = 42
x ==> 42

jshell> x * x
$3 ==> 1764

jshell> /exit
|  Goodbye
```

Using `jshell` can be a great way to learn Java and to experiment with its features. I won't give any more detailed information about it in this book, but you can learn more at

https://docs.oracle.com/en/java/javase/17/jshell/introduction-jshell.html

or you can use the `/help` command inside jshell to learn more about it.

2.6.7 JavaFX on the Command Line

Version 7 of this textbook used Swing for GUI programming. When I began work on Version 8, JavaFX had been added to the standard JDK, and it seemed to be the furture of Java GUIs. So, in Version 8, I made the switch to JavaFX. Not long after that, however, JavaFX was removed from the JDK,

and it is now developed and distributed separately. This has made it somewhat more difficult to use. So for Version 9, I decided to make the book available in two editions, one covering Swing and one covering JavaFX. The only significant differences are in the chapters that cover GUI programming, Chapter 6 and Chapter 13. This edition covers JavaFX. If you also want to learn Swing, you can read those two chapters in the Swing edition.

To use JavaFX on the command line or in Eclipse, you have to download it onto your computer. You can find download links on this page:

https://gluonhq.com/products/javafx/

You need to download a JavaFX "SDK"—**not** "jmods" or "Monocle SDK"—that is appropriate for your operating system (Linux, MacOS, or Windows) and for the JDK that you are using. You should get the version number that matches the version of the JDK; that probably means JavaFX version 17.0.2. (Only the major version number, 17, has to match.) Furthermore, you need to select the "Architecture," which refers to the CPU type. The architecture should match the architecture used by your JDK. If you are using an "x64" JDK from Adoptium or if you are using an "x86_64" Eclipse package, then you need JavaFX for the x64 architecture. If you are using an "aarch64" JDK from Adoptium or an "AArch64" package from Eclipse, then you need the aarch64 architecture.

When you download the JavaFX SDK, it will be in the form of a compressed archive file. You will need to extract the contents of the archive. Usually, simply double-clicking the icon of the archive file will either extract the contents or open a program that you can use to extract the contents. You will get a directory with a name something like *javafx-sdk-17.0.2*. You can put the directory anywhere on your computer, but you will need to know where it is located.

JavaFX is a collection of Java classes that can be used for making GUI programs. In this book, it is first used in Section 3.9 and is covered extensively in Chapter 6 and Chapter 13. It is also used in example programs in several other chapters. This subsection explains how to use JavaFX on the command line. It assumes that you have already downloaded the JavaFX SDK, as described above. JavaFX is distributed as a set of "modules." (See

Subsection 4.6.4) The modules are stored in .jar files in the *lib* subdirectory of the JavaFX SDK. When using the `javac` and `java` commands on a program that uses JavaFX, you need to tell the command where to find the JavaFX modules.

The modules are specified for the `javac` and `java` commands using two command options: `--module-path` and `--add-modules`. The first option specifies the directory that contains the JavaFX .jar files, and the second says which modules you actually want to use. For the purposes of this textbook, you can set the value of `--add-modules` to `ALL-MODULE-PATH`, which makes all of JavaFX available to your program. The value of `--module-path` is a path to the *lib* directory that contains the JavaFX .jar files. For example, let's say that the JavaFX SDK directory is named *openjfx-sdk-17.0.2* and that it is in my home directory, */home/eck*. Then the full path to the *lib* directory is `/home/eck/openjfx-sdk-17.0.2/lib`, and the `javac` command for compiling JavaFX programs would be:

```
javac --module-path=/home/eck/openjfx-sdk-17.0.2/lib --
add-modules=ALL-MODULE-PATH
```

This would be followed on the same line by the .java files that you want to compile. Exactly the same options would be used with the `java` command to run JavaFX programs. The option `--module-path` can also be abbreviated to `-p`, with no equals sign. So this can also be written

```
javac -p /home/eck/openjfx-sdk-17.0.2/lib --add-
modules=ALL-MODULE-PATH
```

If you don't know the full path to the JavaFX SDK, open a command window and use the *cd* command to move to the SDK's *lib* directory. On Mac or Linux, enter the command *pwd* to print out the full path of the working directory. On windows, use the command *cd*, with no directory specified, to print out the path. Use the output from that command as the value for the `--module-path`. On windows, a typical `java` command for use with javafx might look something like this:

```
java -p C:\Users\eck\openjfx-sdk-17.0.2\lib --add-
modules=ALL-MODULE-PATH
```

If the path name includes a space, or certain other special characters, it must be enclosed in quotation marks. For example,

```
java -p "C:\JavaFX Support\openjfx-sdk-17.0.2\lib" --add-modules=ALL-MODULE-PATH
```

Of course, this is very verbose, and it would be nice not to have to retype it all the time. On MacOS or Linux, it is easy to define **aliases**, which are shortcuts for long commands. On my computer, I use an alias to define a *jfxc* command for compiling JavaFX programs. The alias is defined as follows:

```
alias jfxc='javac -p /home/eck/javafx-sdk-17.0.2/lib --add-modules=ALL-MODULE-PATH'
```

Similarly, I defined an alias for running JavaFX programs:

```
alias jfx='java -p /home/eck/javafx-sdk-17.0.2/lib --add-modules=ALL-MODULE-PATH'
```

This lets me compile and run JavaFX programs using commands such as

```
jfxc MyJavaFXProgram.java
jfx MyJavaFXProgram
```

To make the alias definitions permanent on my Linux computer, I added them to a file named *.bashrc*. On MacOS, I would put them in a file named *.zshrc* (for MacOS 10.15 and later) or in a file named *.bash_profile* (for earlier versions of MacOS). That file must be placed in your home directory. The file might or might not already exist; if it doesn't exist, you can create it. The file is executed whenever you open a Terminal window. (In particular, changes do not become effective until you open a new Terminal.) Note that the file name begins with a period, which makes it a "hidden file." That means that it won't show up in a usual directory listing or file browser, but you should be able to set your file browser to show hidden files.

Unfortunately, Windows currently does not have an equivalent of a .bashrc or .zshrc for its *cmd* command window. One option is to make a batch script file to run the command. For compilation, you could create a file named *jfxc.bat* containing just one line similar to

```
javac -p C:\Users\eck\javafx-sdk-17.0.2\lib --add-
modules=ALL-MODULE-PATH %*
```

but, of course, using the appropriate JavaFX location for your own computer. The "`%*`" at the end represents the inputs to the `javac` command. The file can be in the current directory or somewhere on the system path, such as the JDK *bin* directory. Then you can use `jfxc` as a command for compiling JavaFX programs. You can handle the `java` command with a similar .bat file.

2.6.8 Using JavaFX in Eclipse

To compile and run JavaFX programs in Eclipse, you need to configure your Eclipse workspace to support it. My goal here is **not** to present the most "correct" or general way to configure Eclipse for JavaFX; I just want to make it possible to easily work with basic JavaFX programs like the ones that come with this textbook. I will discuss a one-time configuration of an Eclipse workspace that seems to work reliably for all the versions of Eclipse in which I have tried it.

The first step is to download a JavaFX SDK from the JavaFX download site, as discussed in the previous subsection.

To begin, open the Eclipse preferences, using the "Preferences" command (which is in the "Windows" menu on Linux and Windows and in the "Eclipse" menu on MacOS). Expand the Java section in the list on the left, by clicking the small triangle or plus sign next to the word "Java." Click on "Installed JREs." You will see a list of the Java environments that Eclipse knows about. The list should include at least the JRE that is included in Eclipse, and it might also include other JREs that have been installed. You can click on a JRE to select it and click the "Duplicate" button to make a copy. You can then add JavaFX support to the copy. (You could also just "Edit" the original JRE, or "Add" a completely new configuration. I recommend using "Duplicate", but if you add a new configuration, you would have the freedom to use a different JDK that you have downloaded, along with a matching JavaFX.)

When you "Duplicate" an existing JRE, you should see a dialog box similar to the following. The "JRE home" and "JRE name" will already be filled in, and there will be just one entry under "JRE system libraries".

You should change the "JRE name" to something that indicates that it supports JavaFX. This is just so that you will recognize the name when you create a new project in Eclipse. But the main thing is to add the JavaFX configuration.

Remember that to use JavaFX, you need to make it available to your program both at compile time and at run time. The first step in the configuration is to make it available at compile time. To do that, you want to add the JavaFX .jar files to the JRE system libraries: Click the "Add External JARs" button, and navigate to the *lib* directory in the JavaFX SDK that you downloaded. You should see the JavaFX .jar files. Select them all, and click "OK". They should appear in the "JRE system libraries" list.

The second step is to configure the JavaFX source file. This step is optional, but it will allow Eclipse to find the documentation for JavaFX, which can be very useful when you are writing programs. For this step, make sure that all of the JavaFX jar files are selected in the list of .jar files, as shown in the above illustration. Click the "Source Attachment" button. In the "Source Attachment Configuration" dialog box, select "External Location", and click "External File". Select the file *src.zip* from the JavaFX SDK directory, and click "OK". That's all there is to it.

Finally, you must make JavaFX available to your programs at run time. To do that, you need to fill in the "Default VM arguments" input box by typing in the JavaFX command line options for the *java* command. The input box contains options that will be added to the *java* command every time you run a program in Eclipse using this JRE. The `-p` (or `--module-path`) and `--add-modules` options that are needed were discussed in the previous subsection. Examples appropriate for my own computer are shown in the above illustration. The value of the `-p` option is the location of the JavaFX *lib* directory. It should be exactly the same as the beginning of the paths for the JavaFX .jar files as shown in the "JRE system libraries" list. (But enclose the value in quotation marks if it includes spaces or other special characters.) The value for `--add-modules` can be `ALL-MODULE-PATH`, as shown in the illustration.

When everything is set up, click "Finish" and then click "Apply and Close" in the main "Preferences" dialog box. When you create a new Java project in the workspace, make sure that the JRE that you set up to use JavaFX is selected in the project creation dialog box. Look in the "JRE" section of the dialog box, select "Use project-specific JRE," and select the JRE with JavaFX from the popup menu. You should then be able to use JavaFX in that project. If not, check your JRE configuration. You can "Edit" it in the same Java preferences where you set it up in the first place. If you can't compile JavaFX programs, make sure that you are using a JavaFX SDK with the same major version number as the built-in JDK in Eclipse (JavaFX 17 for the March 2022 version of Eclipse, for example). If you can compile JavaFX programs but not run them, check the "Default VM arguments" and check that you have the JavaFX SDK for the correct operating system and architecture.

Programming Exercises for Chapter 2

Exercise 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:

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

See the Solution

Exercise 2.2:

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 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
```

See the Solution

Exercise 2.3:

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!".

See the Solution

Exercise 2.4:

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.

See the Solution

Exercise 2.5:

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.

See the Solution

Exercise 2.6:

This exercise asks you to write a program that tests some of the built-in subroutines for working with *Strings*. The program should ask the user to enter their first name and their last name, separated by a space. Read the user's response using `TextIO.getln()`. Break the input string up into two strings, one containing the first name and one containing the last name. You can do that by using the `indexOf()` subroutine to find the position of the space, and then using `substring()` to extract each of the two names. Also output the number of characters in each name, and output the user's initials. (The initials are the first letter of the first name together with the first letter of the last name.) A sample run of the program should look something like this:

```
Please enter your first name and last name, separated by a
space.
? Mary Smith
Your first name is Mary, which has 4 characters
Your last name is Smith, which has 5 characters
Your initials are MS
```

See the Solution

Exercise 2.7:

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 that 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.

See the Solution

Quiz on Chapter 2

Question 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.

Question 2:

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

Question 3:

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

Question 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?

Question 5:

Give the meaning of each of the following Java operators:

```
a)      ++

b)      &&

c)      !=
```

Question 6:

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

Question 7:

What is meant by *precedence* of operators?

Question 8:

What is a *literal*?

Question 9:

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

Question 10:

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

Question 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` ?

Question 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?

Question 13:

What is the purpose of an import directive, such as `import textio.TextIO` or `import java.util.Scanner`?

Question 14:

Write a complete program that asks the user to enter the number of "widgets" they want to buy and the cost per widget. The program should then output the total cost for all the widgets. Use `System.out.printf` to print the cost, with two digits after the decimal point. You do not need to include any comments in the program.

See the Answers

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.

Program complexity can be seen not just in control structures but also in **data structures**. A data structure is an organized collection of data, chunked together so that it can be treated as a unit. Section 3.8 in this chapter includes an introduction to one of the most common data structures: **arrays**.

The 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. Finally, Section 3.9 is a very brief first look at GUI programming.

Contents of Chapter 3:

- Section 1: Blocks, Loops, and Branches
- Section 2: Algorithm Development
- Section 3: The while and do..while Statements

- Section 4: The for Statement
- Section 5: The if Statement
- Section 6: The switch Statement
- Section 7: Introduction to Exceptions and try..catch
- Section 8: Introduction to Arrays
- Section 9: Introduction to GUI Programming
- Programming Exercises
- Quiz on This Chapter

Section 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 it is 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 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 (at least conceptually). 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, she was making a joke about the way that computers mindlessly follow instructions.)) By the way, if you are working on the command line and need to stop a program that has gotten into an infinite loop, you should be able to do so with Control-C, that is, hold down the Control key and press the C key.

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, most `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` when the computer evaluates it; 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 when the computer evaluates the expression "`number < 6`" for the first time, 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:

```
import textio.TextIO;

/**
 *  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 )
    statement1
else
    statement2
```

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**. In either case, the computer then continues with whatever follows the `if` statement in the program.

Sometimes, novice programmers confuse `while` statements with simple `if` statements (with no `else` part), although their meanings are quite different. The **statement** in an `if` is executed at most once, while the **statement** in a `while` can be executed any number of times. It can be helpful to look at diagrams of the flow of control for `while` and simple `if` statements:

In these diagrams, the arrows represent the flow of time as the statement is executed. Control enters the diagram at the top and leaves at the bottom. Similarly, a flow control diagram for an `if..else` statement makes it clear that exactly one of the two nested statements is executed:

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!

3.1.4 Control Abstractiont

Control structures such as `if` statements and `while` loops allow you to control processes that go on inside the CPU of a computer. But as to what actually goes on in the CPU, there is no such thing as an `if` statements or a `while` loops. As discussed in Section 1.1, a CPU can only execute simple machine language instructions. Loops and branches must be implemented in machine language using jump instructions, which change the value stored in the program counter register, and conditional jump instructions, which might or might not change the program counter, depending on the result of a simple test.

Implementing Java control structures in machine language can get rather complicated. Fortunately, you don't need to know how to do it. You can use Java control structures without knowing how they are implemented in machine language. In fact, you don't need to know anything about machine language or how CPUs work. This is an example of **abstraction**.

Abstraction is a central concept in computer science. Abstraction allows you to work with something without understanding how it works in detail. It allows you to work on a "higher level." `If` statements and `while` loops are examples of **control abstractions**. They allow you to work with a high-level programming language like Java, rather than with very low-level machine language.

You will encounter other kinds of abstraction later in this book. In fact, you've already encountered one: A variable is a basic example of **data abstraction**. A variable lets you use a name to work with a data value of a certain type. The computer has to keep track of the numerical address of that

data in memory, how many bytes of memory the data occupies, and how to interpret the bits that are stored in that memory to represent the value. The variable is an abstraction that lets you avoid all that detail.

3.1.5 Definite Assignment

I will finish this introduction to control structures with a somewhat technical issue that you might not fully understand the first time you encounter it. Consider the following two code segments, which seem to be entirely equivalent:

```
int y;
if (x < 0) {
    y = 1;
}
else {
    y = 2;
}
System.out.println(y);
```

```
int y;
if (x < 0) {
    y = 1;
}
if (x >= 0) {
    y = 2;
}
System.out.println(y);
```

In the version on the left, `y` is assigned the value 1 if `x < 0` and it is assigned the value 2 otherwise, that is, if `x >= 0`. Exactly the same is true of the version on the right. However, there is a subtle difference. In fact, the Java compiler will report an error for the `System.out.println` statement in the code on the right, while the code on the left is perfectly fine!

The problem is that in the code on the right, the computer can't tell that the variable `y` has definitely been assigned a value. When an `if` statement has no `else` part, the statement inside the `if` might or might not be executed, depending on the value of the condition. The compiler can't tell whether it will be executed or not, since the condition will only be evaluated when the program is running. For the code on the right above, as far as the compiler is concerned, it is possible that **neither** statement, `y = 1` or `y = 2`, will be evaluated, so it is possible that the output statement is trying to print an undefined value. The compiler considers this to be an error. The value of a variable can only be used if the compiler can **verify** that the variable will have been assigned a value at that point when the program is running. This is called **definite assignment**. (It doesn't matter that **you** can tell that `y` will

always be assigned a value in this example. The question is whether the compiler can tell.)

Note that in the code on the left above, `y` is definitely assigned a value, since in an `if..else` statement, one of the two alternatives will be executed no matter what the value of the condition in the `if`. It is important that you understand that there is a difference between an `if..else` statement and a pair of plain `if` statements. Here is another pair of code segments that might seem to do the same thing, but don't. What's the value of `x` after each code segment is executed?

```
int x;
x = -1;
if (x < 0)
    x = 1;
else
    x = 2;
```

```
int x;
x = -1;
if (x < 0)
    x = 1;
if (x >= 0)
    x = 2;
```

After the code on the left is executed, `x` is 1; after the code on the right, `x` is 2. If you don't believe this, work though the code step-by-step, doing exactly what the computer does when it executes each step. The key point is that for the code on the right, both assignment statements, `x = 1` and `x = 2`, are executed. Make sure that you understand why.

Section 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 always terminates after a finite number of steps. We don't want to count procedures that might 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 what the task needs to accomplish in the end. 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 an actual 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 more likely just 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. This is sometimes called a **counting loop**. 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 used 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. The complete sequence is: 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. The variable must be set to zero once, **before** the loop starts, and it must be incremented within the loop. (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. After the while loop ends, we can be absolutely sure that `N` is a positive number.)

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.

```
import textio.TextIO;

/**
 * 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


         System.out.print("Starting point for sequence:
");
         N = TextIO.getlnInt();
         while (N <= 0) {
            System.out.print(
                   "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;
             System.out.println(N);
             counter = counter + 1;
         }

         System.out.println();
         System.out.print("There were ");
         System.out.print(counter);
         System.out.println(" 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:

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

Second, there is the question of why this problem might be 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! (Note: This discussion really applies to integers, not to values of type int! That is, it 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. So the Java program does not compute the correct 3N+1 sequence if N becomes too large. 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.

Section 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 full source code for the program (with comments added, of course!):

```
import textio.TextIO;

/**
 * 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. */

      System.out.print("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.
         System.out.print("Enter your next positive
integer, or 0 to end: ");
         inputNumber = TextIO.getlnInt();
      }

      /* Display the result. */

      if (count == 0) {
         System.out.println("You didn't enter any data!");
      }
      else {
         average = ((double)sum) / count;
         System.out.println();
         System.out.println("You entered " + count + "
positive integers.");
         System.out.printf("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" (among other acceptable

responses). "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();
   System.out.print("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!
   System.out.print("Enter a positive number: ");
   N = TextIO.getlnInt();
   if (N > 0)   // the input value is OK, so jump out of
loop
```

```
      break;
   System.out.println("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;  // so nothingInCommon
is actually 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 only if the `if` statement is nested inside a loop or inside a `switch` statement. 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. The same consideration applies to `continue` statements inside `if`s.

Section 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 (and I know of at least one person who **only** uses `for` 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 (including the first execution), 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 simple statement, but is usually an increment, a decrement, or an assignment statement. Any of the three parts can be empty, but the two semicolons are required in any case. 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)`".) Here's a flow control diagram for a `for` statement:

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 example given above, the loop control variable was `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 except for a variable declaration, 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. (This is a correct solution, but there are much more efficient ways to perform this task.) 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 ten 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 10000000, the program outputs a '.' and resets the counter to zero so that we can start counting the next group of ten 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 10000000:
        print out a '.'
```

```
        Reset numberTested to 0
Output the count
```

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

```
import textio.TextIO;

/**
 * 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 10000000, 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
```

```
10000000 tests. */

      divisorCount = 0;
      numberTested = 0;

      for (testDivisor = 1; testDivisor <= N;
testDivisor++) {
         if ( N % testDivisor == 0 )
            divisorCount++;
         numberTested++;
         if (numberTested == 10000000) {
            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 other, simpler 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, and it is a style rule that excessive nesting should be avoided.

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 `if` statement inside the `for` loop needs still more thought before we can write the program. 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:

```
import textio.TextIO;

/**
 * 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.

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

      str = str.toUpperCase();
```

```
      count = 0;
      System.out.println("Your input contains the
following letters:");
      System.out.println();
      System.out.print("   ");
      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) ) {
                  System.out.print(letter);
                  System.out.print(' ');
                  count++;
                  break;
              }
          }
      }

      System.out.println();
      System.out.println();
      System.out.println("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.

Section 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 Multiway Branching

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 **statements**. 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 multiway branches with any number of cases:

```
if (test-1)
     statement-1
```

```
else if (test-2)
     statement-2
else if (test-3)
     statement-3
  .
  . // (more cases)
  .
else if (test-N)
     statement-N
else
     statement-(N+1)
```

The computer evaluates the tests, which are 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 multiway branch because one and 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. It might be useful to look at a flow control diagram for the general "if..else if" statement shown above:

3.5.3 If Statement Examples

As an example of using `if` statements, let's 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 example, 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 under-specified. 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 program that, I just had 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:

```
import textio.TextIO;

/**
 * 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.

       System.out.println("""
                Enter measurements in inches, feet, yards,
or miles.
                For example:  1 inch    17 feet    2.73
miles
                You can use abbreviations:   in   ft  yd
mi
                I will convert your input into the other
units
                of measure.
                """);

       while (true) {

          /* Get the user's input, and convert units to
```

```
lower case. */

          System.out.print("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 units
to lower case

          /* 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 {
              System.out.println("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. */

            System.out.printf("""
                      That's equivalent to:
                      %14.5g inches
                      %14.5g feet
                      %14.5g yards
                      %14.5g miles
                      """, inches, feet, yards, miles);

            System.out.println();

        } // end while

        System.out.println();
        System.out.println("OK!  Bye for now.");

    } // end main()

  } // end class LengthConverter
```

(Note that this program uses text blocks for multiline outputs (see Subsection 2.3.4; text blocks require Java 17). It also 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 `%14.5g`, 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 `%14.5f`, 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 (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 empty 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. The `for` loop just does nothing, ten times!

Section 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 much less often than the `if` statement, but it is sometimes useful for expressing a certain type of multiway 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. It can be *String*. Or it can be an enum type (see Subsection 2.3.5 for an introduction to enum types). In particular, note that the expression **cannot** be a double or float value.

Java has two different syntaxes for `switch` statements. The original switch syntax, which like other Java control structures was modeled on the C programming language, is error-prone and kind-of ugly. The new syntax, which requires Java 17, is an improvement. You still need to know the traditional syntax, since it is used in a lot of existing code, but my advice would be to use the new syntax when you write new code. We look at the new syntax first, and will cover the traditional syntax at the end of this section.

The positions within a switch statement to which it can jump are marked with **case labels** that take the form: "case **constantList**". The **constantList** here consists of one or more literals of the same type as the expression in the `switch`, separated by commas. For example:

```
        case 2, 4, 8
or
        case "Paper"
```

The case label is followed by `->`, that is by a symbol made up of a hyphen and a greater-than character, and then by a single statement. The statement can be a subroutine call statement, a `throw` statement, or a block statement, containing several nested statements. A switch statement can also, optionally, have one jump point labeled with `default` instead of with a case label. The syntax for the statement can be specified as follows, noting that there can be at most one `default` case and that all the constants in the case labels must be different:

```
switch ( expression ) {
    case-label-or-default -> statement
    case-label-or-default -> statement
        .
        .
        .
    case-label-or-default -> statement
}
```

When the computer executes this `switch` statement, it evaluates the **expression**. If the value is one of the constants in a case label, the computer executes the statement that follows that case label, and then jumps out of the `switch` statement. If the value of the expression does not match any of the case constants, then the computer looks for a `default` case, and if one is present, executes the statement that follows it.

It is probably easiest to look at an example. This is not a useful example, but it should be easy to follow:

```
switch ( N ) {   // (Assume N is an integer variable.)
   case 1 -> System.out.println("The number is 1.");
   case 2, 4, 8 -> {
      System.out.println("The number is 2, 4, or 8.");
      System.out.println("(That's a power of 2!)");
   }
   case 3, 6, 9 -> {
      System.out.println("The number is 3, 6, or 9.");
      System.out.println("(That's a multiple of 3!)");
   }
```

```
   case 5 -> System.out.println("The number is 5.");
   default ->
      System.out.println("The number is 7 or is outside
the range 1 to 9.");
}
```

The braces, `{` and `}`, in this example are required to group multiple statements into a single block statement. Braces could also be added to the other cases, but are not required there. This `switch` statement has exactly the same effect as the following multiway if statement:

```
if ( N == 1 ) {
    System.out.println("The number is 1.");
}
else if ( N == 2 || N == 4 || N == 8 ) {
    System.out.println("The number is 2, 4, or 8.");
    System.out.println("(That's a power of 2!)");
}
else if ( N == 3 || N == 6 || N == 9 ) {
    System.out.println("The number is 3, 6, or 9.");
    System.out.println("(That's a multiple of 3!)");
}
else if ( N == 5 ) {
    System.out.println("The number is 5.");
}
else {
    System.out.println("The number is 7 or is outside the
range 1 to 9.");
}
```

More generally, any `switch` statement could be replaced by a multiway if statement. The `switch` statement can be easier to read. And it might be more efficient since the computer can jump directly to the correct case instead of working through a series of tests to get to the correct case.

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 command-line program, the menu can be presented as a numbered list of options, and the user can choose an option by typing in its number. It can be convenient to use a text block (see Subsection 2.3.4) to present the menu. 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 of options, and get user's selected option
number. */

System.out.println("""
        What unit of measurement does your input use?

                1. inches
                2. feet
                3. yards
                4. miles

        Enter the number of your choice:
        """);

optionNumber = TextIO.getlnInt();

/* Read user's measurement and convert to inches. */

switch ( optionNumber ) {
   case 1 -> {
       System.out.println("Enter the number of inches: ");
       measurement = TextIO.getlnDouble();
       inches = measurement;
   }
   case 2 -> {
       System.out.println("Enter the number of feet: ");
```

```
        measurement = TextIO.getlnDouble();
        inches = measurement * 12;
    }
    case 3 -> {
        System.out.println("Enter the number of yards: ");
        measurement = TextIO.getlnDouble();
        inches = measurement * 36;
    }
    case 4 -> {
        System.out.println("Enter the number of miles: ");
        measurement = TextIO.getlnDouble();
        inches = measurement * 12 * 5280;
     }
    default -> {
        System.out.println("Error!  Illegal option number!
I quit!");
        System.exit(1);
    }

} // end switch

/* Now go on to convert inches to feet, yards, and
miles... */
```

Alternatively, this example could be designed to ask the use to enter the unit of measure as a string, instead of as an option number, and then use that string directly in a `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. */

System.out.println("What unit of measurement does your
input use?");
units = TextIO.getln().toLowerCase();

/* Read user's measurement and convert to inches. */

System.out.print("Enter the number of " + units + ":  ");
measurement = TextIO.getlnDouble();
```

```
switch ( units ) {
   case "inch", "inches", "in" ->  inches = measurement;
   case "foot", "feet", "ft"   ->  inches = measurement *
12;
   case "yard", "yards", "yd"  ->  inches = measurement *
36;
   case "mile", "miles", "mi"  ->  inches = measurement *
12 * 5280;
   default -> {
       System.out.println("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 enum type. In that case, the constants in the `case` labels must be values from the enum type. For example, suppose that the type of the expression is the enumerated type *Season* defined by

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

and that the expression in a `switch` statement is an expression of type *Season*. 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 a quirk in the syntax: when an enum constant is used in a `case` label, only the simple name, such as "`SPRING`" is used, not the full name, such as "`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. For example, assuming that `currentSeason` is a variable of type *Season*, then we could have the `switch` statement:

```
System.out.print("The months in " + currentSeason + " are:
");

switch ( currentSeason ) {
   case WINTER ->   // ( NOT Season.WINTER ! )
```

```
      System.out.println("December, January, February");
   case SPRING ->
      System.out.println("March, April, May");
   case SUMMER ->
      System.out.println("June, July, August");
   case FALL ->
      System.out.println("September, October, November");
}
```

3.6.4 Definite Assignment and switch Statements

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"`, `"Paper"`, `"Scissors"` to `computerMove`, with probability 1/3 for each case:

```
switch ( (int)(3*Math.random()) ) {
   case 0 -> computerMove = "Rock";
   case 1 -> computerMove = "Paper";
   case 2 -> computerMove = "Scissors";
}
```

This `switch` statement is perfectly OK, but suppose that we use it in the following code segment:

```
String computerMove;
switch ( (int)(3*Math.random()) ) {
   case 0 -> computerMove = "Rock";
   case 1 -> computerMove = "Paper";
   case 2 -> computerMove = "Scissors";
}
System.out.println("The computer's move is " +
computerMove);  // ERROR!
```

Now there is a subtle error on the last line! The problem here is due to definite assignment, the idea that the Java compiler must be able to determine that a variable has definitely been assigned a value before its value is used. Definite assignment was introduced in Subsection 3.1.5. In this example, it's true that the three cases in the `switch` cover all the possibilities, but the

compiler is not smart enough to figure that out; it just sees that there is an integer-valued expression in the `switch` but not all possible integer values are covered by the given cases.

A simple solution is to replace the final `case` in the `switch` statement with `default`. With a `default` case, all possible values of the expression in the `switch` are certainly covered, and the compiler knows that `computerMove` is definitely assigned a value:

```
String computerMove;
switch ( (int)(3*Math.random()) ) {
   case 0 -> computerMove = "Rock";
   case 1 -> computerMove = "Paper";
   default -> computerMove = "Scissors";
}
System.out.println("The computer's move is " +
computerMove);  // OK!
```

3.6.5 Switch Expressions

Often, the whole purpose of a `switch` statement is to assign a value to a variable, where the value that is to be assigned depends on the value of the expression in the `switch` statement. For example, this is true for the `switch` statement in the previous subsection.

This type of thing can be handled more elegantly by using a **switch expression** instead of a `switch` statement. Like any expression, a `switch` expression computes and returns a single value. The syntax is similar to a `switch` statement, but instead of a statement in each `case`, there is an expression (or a `throw` statement). For example,

```
String computerMove = switch ( (int)(3*Math.random()) ) {
        case 1 -> "Rock";
        case 2 -> "Paper";
        default -> "Scissors";
    };
```

This is a single assignment statement, where the value to be assigned to `computerMove` is computed by the `switch` expression. The semicolon on the

last line is the semicolon that is required at the end of the assignment statement; it is not part of the `switch` expression.

A `switch` expression must always compute a value (or throw an exception) and therefore will usually have a `default` case. The expression in a `case` can be replaced by a block containing several statements; the value for that `case` should then be specified by a `yield` statement, which takes a form such as "`yield 42;`". The value after the word `yield` is then returned as the value of the `switch` expression.

Of course, `switch` expressions are not limited to assignment statements. You can use a `switch` expression any place where any other kind of expression could be used, such as in an output statement or as part of a larger expression.

3.6.6 The Traditional switch Statement

The older traditional syntax for `switch` statements is more complicated, but you should be aware of it since it has been widely used in Java and in other programming languages. As it is most often used, the traditional `switch` 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
```

Note that in the traditional syntax, only one constant is allowed in a case label (but Java 17 allows a comma-separated list of constants here). A case label can be followed by any number of statements. This traditional syntax uses a colon after each case label, rather than `->`. The `default` case is optional.

To execute this `switch` statement, the computer will evaluate the **expression** and jump to the case label that contains that constant, if there is one, or to the default case if not. The `break` statements in this `switch` are not actually required by the syntax of the `switch` statement. The effect of a `break` is to make the computer jump past the end of the switch statement, skipping over all the remaining cases. 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 called "fall through"; it is rarely what you want, and it is a common source of bugs. However, it is legal and is even occasionally useful.

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 how our first example `switch` statement would be written using the traditional syntax:

```
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.");
}
```

Section 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 and incomplete look the `try..catch` statement, leaving out a lot of the rather complex syntax of this statement. Error handling is a complex topic, which we will return to in Section 8.3, and we will cover the full syntax of `try..catch` at that time.

3.7.1 Exceptions

The term **exception** is used to refer to the type of event 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 usually 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.

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 simplified form, the syntax for a `try..catch` statement can be:

```
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 `try..catch` statement, it executes **statements-1**, the statements inside the `try` part. If no exception 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 from the point where the exception occurs to the `catch` part and executes **statements-2**, skipping any remaining statements in **statements-1**. 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.

An exception is represented by an object. During the execution of **statements-2**, the **variable-name** represents that exception object, so that you can, for example, print it out. The exception object contains information about the cause of the exception. This includes an error message, which will be displayed if you print out the exception object.

After 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.

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:

```
double x;
try {
   x = Double.parseDouble(str);
   System.out.println( "The number is " + x );
}
catch ( NumberFormatException e ) {
   System.out.println( "Not a legal number." );
   x = Double.NaN;
}
```

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. (In this example, I set `x` to be the value `Double.NaN` when

an exception occurs. `Double.NaN` is the special "not-a-number" value for type double.)

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.

Suppose, for example, we want a program that will find the average of a sequence of real numbers entered by the user, and we want the user to signal the end of the sequence by entering a blank line. (This is similar to the sample program ComputeAverage.java from Section 3.3, but in that program the user entered a zero to signal end-of-input.) If we use `TextIO.getlnInt()` to read the user's input, we will have no way of detecting the blank line, since that function simply skips over blank lines. A solution is to use `TextIO.getln()` to read the user's input. This allows us to detect a blank input line, and we can convert non-blank inputs to numbers using `Double.parseDouble`. And we can use `try..catch` to avoid crashing the program when the user's input is not a legal number. In this example, it makes sense to simply print an error message, ignore the bad input, and let the program continue. Here's the program:

```
import textio.TextIO;

public class ComputeAverage2 {

   public static void main(String[] args) {
       String str;     // The user's input.
       double number;  // The input converted into a
number.
       double total;   // The total of all numbers
entered.
       double avg;     // The average of the numbers.
       int count;      // The number of numbers entered.
       total = 0;
       count = 0;
       System.out.println("Enter your numbers, press
return to end.");
       while (true) {
          System.out.print("? ");
          str = TextIO.getln();
          if (str.equals("")) {
```

```
                break; // Exit the loop, since the input line
was blank.
            }
            try {
                number = Double.parseDouble(str);
                // If an error occurs, the next 2 lines are
skipped!
                total = total + number;
                count = count + 1;
            }
            catch (NumberFormatException e) {
                System.out.println("Not a legal number!  Try
again.");
            }
        }
        avg = total/count;
        System.out.printf("The average of %d numbers is
%1.6g%n", count, avg);
    }

}
```

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 2.4.4.) 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*.

As an example, we will look at yet another number-averaging program. In this case, we will read the numbers from a file. Assume that the 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 when it occurs. 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:

```
import textio.TextIO;

/**
 * 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.  (The user can choose to end the
program by
 * typing Control-C.)
 */
public class AverageNumbersFromFile {

   public static void main(String[] args) {

      while (true) {
         String fileName;  // The name of the file, to be
input by the user.
```

```
         System.out.print("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 ) {
            System.out.println("Can't read from the file
\"" + fileName + "\".");
            System.out.println("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.

      System.out.println();
```

```
      System.out.println("Number of data values read: " +
count);
      System.out.println("The sum of the data values: " +
sum);
      if ( count == 0 )
         System.out.println("Can't compute an average of 0
values.");
      else
         System.out.println("The average of the values:  "
+ (sum/count));

   }

}
```

Section 3.8

Introduction to Arrays

IN PREVIOUS SECTIONS OF THIS CHAPTER, we have already covered all of Java's control structures. But before moving on to the next chapter, we will take preliminary looks at two additional topics that are at least somewhat related to control structures.

This section is an introduction to arrays. Arrays are a basic and very commonly used data structure, and array processing is often an exercise in using control structures. The next section will introduce computer graphics and will allow you to apply what you know about control structures in another context.

3.8.1 Creating and Using Arrays

A **data structure** consists of a number of data items chunked together so that they can be treated as a unit. An **array** is a data structure in which the items are arranged as a numbered sequence, so that each individual item can be referred to by its position number. In Java—but not in some other programming languages—all the items must be of the same type, and the numbering always starts at zero. You will need to learn several new terms to talk about arrays: The number of items in an array is called the **length** of the array. The type of the individual items in an array is called the **base type** of the array. And the position number of an item in an array is called the **index** of that item.

Suppose that you want to write a program that will process the names of, say, one thousand people. You will need a way to deal with all that data. Before you knew about arrays, you might have thought that the program would need a thousand variables to hold the thousand names, and if you wanted to print out all the names, you would need a thousand print statements. Clearly, that

would be ridiculous! In reality, you can put all the names into an array. The array is represented by a single variable, but it holds the entire list of names. The length of the array would be 1000, since there are 1000 individual names. The base type of the array would be *String* since the items in the array are strings. The first name would be at index 0 in the array, the second name at index 1, and so on, up to the thousandth name at index 999.

The base type of an array can be any Java type, but for now, we will stick to arrays whose base type is *String* or one of the eight primitive types. 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), just as a variable can hold a value. The value can be changed at any time, just as the value of a variable can be changed. 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.

As I mentioned above, when you use an array in a program, you can use a variable to refer to the array as a whole. But you often need to refer to the individual elements of the array. The name for an element of an array is based on the name for the array and the index number of the element. The syntax for referring to an element looks, for example, like this: `namelist[7]`. Here, `namelist` is the variable that names the array as a whole, and `namelist[7]` refers to the element at index 7 in that array. That is, to refer to an element of an array, you use the array name, followed by element index enclosed in square brackets. An element name of this form can be used like any other variable: You can assign a value to it, print it out, use it in an expression, and so on.

An array also contains a kind of variable representing its length. For example, you can refer to the length of the array `namelist` as

`namelist.length`. However, you cannot assign a value to `namelist.length`, since the length of an array cannot be changed.

Before you can use a variable to refer to an array, that variable must be declared, and it must have a type. For an array of *Strings*, for example, the type for the array variable would be `String[]`, and for an array of ints, it would be `int[]`. In general, an array type consists of the base type of the array followed by a pair of empty square brackets. Array types can be used to declare variables; for example,

```
String[] namelist;
int[] A;
double[] prices;
```

and variables declared in this way can refer to arrays. However, declaring a variable does not make the actual array. Like all variables, an array variable has to be assigned a value before it can be used. In this case, the value is an array. Arrays have to be created using a special syntax. (The syntax is related to the fact that arrays in Java are actually objects, but that doesn't need to concern us here.) Arrays are created with an operator named **new**. Here are some examples:

```
namelist = new String[1000];
A = new int[5];
prices = new double[100];
```

The general syntax is

```
array-variable = new base-type[array-length];
```

The length of the array can be given as either an integer or an integer-valued expression. For example, after the assignment statement "`A = new int[5];`", `A` is an array containing the five integer elements `A[0]`, `A[1]`, `A[2]`, `A[3]`, and `A[4]`. Also, `A.length` would have the value 5. It's useful to have a picture in mind:

When you create an array of int, each element of the array is automatically initialized to zero. Any array of numbers is filled with zeros when it is created. An array of boolean is filled with the value `false`. And an array of char is filled with the character that has Unicode code number zero. (For an array of *String*, the initial value is `null`, a special value used for objects that we won't encounter officially until Section 5.1.)

3.8.2 Arrays and For Loops

A lot of the real power of arrays comes from the fact that the index of an element can be given by an integer variable or even an integer-valued expression. For example, if `list` is an array and `i` is a variable of type int, then you can use `list[i]` and even `list[2*i+1]` as variable names. The meaning of `list[i]` depends on the value of `i`. This becomes especially useful when we want to process all the elements of an array, since that can be done with a `for` loop. For example, to print out all the items in an array, `list`, we can just write

```
int i;  // the array index
for (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. If the length of the list is 5, then 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.

Let's look at a few more examples. Suppose that `A` is an array of double, and we want to find the average of all the elements of the array. We can use a `for` loop to add up the numbers, and then divide by the length of the array to get the average:

```
double total;    // The sum of the numbers in the array.
double average;  // The average of the numbers.
int i;  // The array index.
total = 0;
for ( i = 0; i < A.length; i++ ) {
    total = total + A[i];  // Add element number i to the
total.
}
average = total / A.length;  // A.length is the number of
items
```

Another typical problem is to find the largest number in the array `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;  // The largest number seen so far.
max = A[0];   // At first, the largest number seen is
A[0].
int i;
for ( i = 1; i < A.length; i++ ) {
    if (A[i] > max) {
       max = A[i];
    }
}
// at this point, max is the largest item in A
```

Sometimes, you only want to process some elements of the array. In that case, you can use an `if` statement inside the `for` loop to decide whether or not to process a given element. Let's look at the problem of averaging the elements of an array, but this time, suppose that we only want to average the non-zero elements. In this case, the number of items that we add up can be less than the

length of the array, so we will need to keep a count of the number of items added to the sum:

```
double total;    // The sum of the non-zero numbers in the
array.
int count;       // The number of non-zero numbers.
double average;  // The average of the non-zero numbers.
int i;
total = 0;
count = 0;
for ( i = 0; i < A.length; i++ ) {
    if ( A[i] != 0 ) {  // Only process non-zero elements!
        total = total + A[i];  // Add element to the total
        count = count + 1;     //      and count it.
    }
}
if (count == 0) {
    System.out.println("There were no non-zero
elements.");
}
else {
    average = total / count;  // Divide by number of items
    System.out.printf("Average of %d elements is %1.5g%n",
                            count, average);
}
```

3.8.3 Random Access

So far, 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];
```

For this problem, 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 are false. (Remember that this is done automatically when the array is created.) 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. On the other hand, 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 program that carries out the simulated experiment (of course, in the program, 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.
 */
public class BirthdayProblem {

   public static void main(String[] args) {

       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++;

          System.out.printf("Person %d has birthday number
%d%n", count, birthday);

          if ( used[birthday] ) {
                // This day was found before; it's a
duplicate.  We are done.
             break;
          }

          used[birthday] = true;

       } // end while

       System.out.println();
       System.out.println("A duplicate birthday was found
after "
                                                + count + "
tries.");
   }

} // end class BirthdayProblem
```

You should study the program to understand how it works and how it uses the array. Also, try it out! You will probably find that a duplicate birthday tends to occur sooner than you expect.

3.8.4 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 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. Each time a number is stored in the array, we have to count it; that is, the value of the counter variable must be incremented by one. One question is, when we add a new item to the array, where do we put it? Well, if the number of items is `count`, then they would be stored in the array in positions number 0, 1, ..., (count-1). The next open spot would be position number `count`, so that's where we should put the new item.

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. Assume that an input value equal to zero marks the end of the data. (This is, at least, a processing task that requires that the numbers be saved in an array. Note 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.)

```
import textio.TextIO;
```

```
public class ReverseInputNumbers {

   public static void main(String[] args) {

     int[] numbers;  // An array for storing the input
values.
     int count;      // The number of numbers saved in the
array.
     int num;        // One of the numbers input by the
user.
     int i;          // for-loop variable.

     numbers = new int[100];   // Space for 100 ints.
     count = 0;                // No numbers have been
saved yet.

     System.out.println("Enter up to 100 positive
integers; enter 0 to end.");

     while (true) {   // Get the numbers and put them in
the array.
        System.out.print("? ");
        num = TextIO.getlnInt();
        if (num <= 0) {
              // Zero marks the end of input; we have all
the numbers.
           break;
        }
        numbers[count] = num;  // Put num in position
count.
        count++;  // Count the number
     }

     System.out.println("\nYour numbers in reverse order
are:\n");

     for ( i = count - 1; i >= 0; i-- ) {
         System.out.println( numbers[i] );
     }

   } // end main();

}  // end class ReverseInputNumbers
```

It is especially important to note how the variable `count` 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.

When the time comes to print out the numbers in the array, the last occupied spot in the array is location `count - 1`, so the `for` loop prints out values starting from location `count - 1` and going down to 0. This is also a nice example of processing the elements of an array in reverse order.

You might wonder what would happen in this program if the user tries to input more than 100 numbers. The result would be an error that would crash the program. When the user enters the 101-st number, the program tries to store that number in an array element `number[100]`. However, there is no such array element. There are only 100 items in the array, and the index of the last item is 99. The attempt to use `number[100]` generates an exception of type *ArrayIndexOutOfBoundsException*. Exceptions of this type are a common source of run-time errors in programs that use arrays.

3.8.5 Two-dimensional Arrays

The arrays that we have considered so far are "one-dimensional." This means that the array consists of a sequence of elements that can be thought of as being laid out along a line. It is also possible to have **two-dimensional arrays**, where the elements can be laid out in a rectangular grid. We consider them only briefly here, but will return to the topic in Section 7.6.

In a two-dimensional, or "2D," array, the elements can be arranged in rows and columns. Here, for example, is a 2D array of int that has five rows and seven columns:

	0	1	2	3	4	5	6
0	13	7	33	54	-5	-1	92
1	-3	0	8	42	18	0	67
2	44	78	90	79	-5	72	22
3	43	-6	17	100	1	-12	12
4	2	0	58	58	36	21	87

This 5-by-7 grid contains a total of 35 elements. The rows in a 2D array are numbered 0, 1, 2, ..., up to the number of rows minus one. Similarly, the columns are numbered from zero up to the number of columns minus one.

Each individual element in the array can be picked out by specifying its row number and its column number. (The illustration shown here is not what the array actually looks like in the computer's memory, but it does show the logical structure of the array.)

In Java, the syntax for two-dimensional arrays is similar to the syntax for one-dimensional arrays, except that an extra index is involved, since picking out an element requires both a row number and a column number. For example, if `A` is a 2D array of int, then `A[3][2]` would be the element in row 3, column 2. That would pick out the number 17 in the array shown above. The type for `A` would be given as `int[][]`, with two pairs of empty brackets. To declare the array variable and create the array, you could say,

```
int[][]  A;
A  =  new int[5][7];
```

The second line creates a 2D array with 5 rows and 7 columns. Two-dimensional arrays are often processed using nested `for` loops. For example, the following code segment will print out the elements of `A` in neat columns:

```
int row, col;  // loop-control-variables for accessing
rows and columns in A
for ( row = 0; row < 5; row++ ) {
    for ( col = 0; col < 7; col++ ) {
        System.out.printf( "%7d",  A[row][col] );
    }
    System.out.println();
}
```

The base type of a 2D array can be anything, so you can have arrays of type `double[][]`, `String[][]`, and so on.

There are some natural uses for 2D arrays. For example, a 2D array can be used to store the contents of the board in a game such as chess or checkers. And an example in Subsection 4.7.3 uses a 2D array to hold the colors of a grid of colored squares. But 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 2022. If the stores are numbered from 0 to 24, and if the twelve months from January 2022 through

December 2022 are numbered from 0 to 11, then the profit data could be stored in an array, `profit`, created as follows:

```
double[][]  profit;
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` (where the numbering, remember, starts from zero).

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 2022.
int store, month;  // variables for looping through the
stores and the months
totalProfit = 0;
for ( store = 0; store < 25; store++ ) {
   for ( 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;
int storeNum;
decemberProfit = 0.0;
for ( storeNum = 0; storeNum < 25; storeNum++ ) {
   decemberProfit += profit[storeNum][11];
}
```

Two-dimensional arrays are sometimes useful, but they are much less common than one-dimensional arrays. Java actually allows arrays of even higher dimension, but they are only rarely encountered in practice.

Section 3.9

Introduction to GUI Programming

FOR THE PAST TWO CHAPTERS, you've been learning the sort of programming that is done inside a single subroutine, "programming in the small." In the rest of this book, 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, we see how techniques that you have learned so far can be applied in the context of graphical user interface programming. GUI programs here, and in the rest of this book, are written using JavaFX, a collection of classes that form a "toolkit" for writing GUI programs. All of the classes mentioned in this section are part of JavaFX, and they must be imported into any program that uses them. See Subsection 2.6.7 and Subsection 2.6.8 for information about compiling and running programs that use JavaFX.

When you run a GUI program, it opens one or more windows on your computer screen. As a programmer, you can have complete control over what appears in the window and how the user can interact with it. For our first encounter, we look at one simple example: the ability of a program to display simple shapes like rectangles and lines in the window, with no user interaction. For now, the main point is to take a look at how programming-in-the-small can be used in other contexts besides text-based, command-line-style programs. You will see that a knowledge of programming-in-the-small applies to writing the guts of any subroutine, not just `main()`.

3.9.1 Drawing Shapes

To understand computer graphics, you need to know a little about pixels and coordinate systems. The computer screen is made up of small squares called **pixels**, arranged in rows and columns, usually about 100 pixels per inch. (Many screens now have many more physical pixels per inch. On these

"high-resolution" screens, a "pixel" might refer to a physical pixel, but it might also refer to a "logical pixel," which is a unit of measure somewhere close to 1/100 inch.)

The computer controls the color of the pixels, and drawing is done by changing the colors of individual pixels. Each pixel has a pair of integer coordinates, often called *x* and *y*, that specify the pixel's horizontal and vertical position. When drawing to a rectangular area on the screen, the coordinates of the pixel in the upper left corner of the rectangle are (0,0). The *x* coordinate increases from left to right, and the *y* coordinate increases from top to bottom. Shapes are specified using pixels. For example, a rectangle is specified by the *x* and *y* coordinates of its upper left corner and by its width and height measured in pixels. Here's a picture of a rectangular drawing area, showing the ranges of *x* and *y* coordinates. The "width" and "height" in this picture give the size of the drawing area, in pixels:

Assuming that the drawing area is 800-by-500 pixels, the rectangle in the upper left of the picture would have, approximately, width 200, height 150, and upper left corner at coordinates (50,50).

Drawing in Java is done using a **graphics context**. A graphics context is an object. As an object, it can include subroutines and data. Among the subroutines in a graphics context are routines for drawing basic shapes such as lines, rectangles, ovals, and text. (When text appears on the screen, the characters have to be drawn there by the computer, just like the computer

draws any other shapes.) Among the data in a graphics context are the color and font that are currently selected for drawing. (A font determines the style and size of characters.) One other piece of data in a graphics context is the "drawing surface" on which the drawing is done. Different graphics context objects can draw to different drawing surfaces. For us, the drawing surface will be the content area of a window, not including its border or title bar.

There are two ways to draw a shape in JavaFX: You can **fill** the shape, meaning you can set the color of each of the pixels inside the shape. Or you can **stroke** the shape, meaning that you set the color of the pixels that lie along the border of the shape. Some shapes, such as a line, can only be stroked. A JavaFX graphics context actually keeps track of two separate colors, one used for filling shapes and one used for stroking shapes. Stroking a shape is like dragging a pen along the border of the shape. The properties of that pen (such as its size and whether it produces a solid line or a dashed line) are properties of the graphics context.

A graphics context is represented by a variable. The type for the variable is *GraphicsContext* (just like the type for a string variable is *String*). The variable is often named *g*, but the name of the variable is of course up to the programmer. Here are a few of the subroutines that are available in a graphics context *g*. Note that all numerical parameter values can be of type double.

- `g.setFill(c)` is called to set the color to be used for filling shapes. The parameter, `c` is an object belonging to a class named *Color*. There are many constants representing standard colors that can be used as the parameter in this subroutine. The standard colors range from common colors such as `Color.BLACK`, `Color.WHITE`, `Color.RED`, `Color.GREEN`, `Color.BLUE`, and `Color.YELLOW`, to more exotic color names such as `Color.CORNFLOWERBLUE`. (Later, we will see that it is also possible to create new colors.) For example, if you want to fill shapes with red, you would say "`g.setFill(Color.RED);`". The specified color is used for all subsequent fill operations up until the next time `g.setFill()` is called. Note that previously drawn shapes are **not** affected!

- `g.setStroke(c)` is called to set the color to be used for stroking shapes. It works similarly to `g.setFill`.
- `g.setLineWidth(w)` sets the size of the pen that will be used for subsequent stroke operations, where `w` is measured in pixels.
- `g.strokeLine(x1,y1,x2,y2)` draws a line from the point with coordinates `(x1,y1)` to the point with coordinates `(x2,y2)`. The width of the line is 1, unless a different line width has been set by calling `g.setLineWidth()`, and the color is black unless a different color has been set by calling `g.setStroke()`.
- `g.strokeRect(x,y,w,h)` draws the outline of a rectangle with vertical and horizontal sides. This subroutine draws the outline of the rectangle whose top-left corner is `x` pixels from the left edge of the drawing area and `y` pixels down from the top. The horizontal width of the rectangle is `w` pixels, and the vertical height is `h` pixels. Color and line width are set by calling `g.setStroke()` and `g.setLineWidth()`.
- `g.fillRect(x,y,w,h)` is similar to `g.strokeRect()` except that it fills in the inside of the rectangle instead of drawing an outline, and it uses the color set by `g.setFill()`.
- `g.strokeOval(x,y,w,h)` draws the outline of an oval. The oval just fits inside the rectangle that would be drawn by `g.strokeRect(x,y,w,h)`. To get a circle, use the same values for `w` and for `h`.
- `g.fillOval(x,y,w,h)` is similar to `g.strokeOval()` except that it fills in the inside of the oval instead of drawing an outline.

This is enough information to draw some pictures using Java graphics. To start with something simple, let's say that we want to draw a set of ten parallel lines, something like this:

Let's say that the lines are 200 pixels long and that the distance from each line to the next is 10 pixels, and let's put the start of the first line at the pixel with coordinates (100,50). To draw one line, we just have to call `g.strokeLine(x1,y1,x2,y2)` with appropriate values for the parameters. Now, all the lines start at *x*-coordinate 100, so we can use the constant 100 as the value for `x1`. Since the lines are 200 pixels long, we can use the constant 300 as the value for `x2`. The *y*-coordinates of the lines are different, but we can see that both endpoints of a line have the **same** *y*-coordinates, so we can use a single variable as the value for `y1` and for `y2`. Using `y` as the name of that variable, the command for drawing one of the lines becomes `g.strokeLine(100,y,300,y)`. The value of `y` is 50 for the top line and increases by 10 each time we move down from one line to the next. We just need to make sure that `y` takes on the correct sequence of values. We can use a for loop that counts from 1 to 10:

```
int y;   // y-coordinate for the line
int i;   // loop control variable
y = 50;  // y starts at 50 for the first line
for ( i = 1; i <= 10; i++ ) {
    g.strokeLine( 100, y, 300, y );
    y = y + 10;  // increase y by 10 before drawing the
next line.
}
```

Alternatively, we could use `y` itself as the loop control variable, noting that the value of `y` for the last line is 140:

```
int y;
for ( y = 50; y <= 140; y = y + 10 )
    g.strokeLine( 100, y, 300, y );
```

If we wanted the lines to be blue, we could do that by calling `g.setStroke(Color.BLUE)` **before** drawing them. If we just draw the lines without setting the color, they will be black. If we wanted the lines to be 3 pixels wide, we could call `g.setLineWidth(3)` **before** drawing the lines.

For something a little more complicated, let's draw a large number of randomly colored, randomly positioned, filled circles. Since we only know a few colors, I will randomly select the color to be red, green, blue, or yellow.

That can be done with a simple switch statement, similar to the ones in Section 3.6:

```
switch ( (int)(4*Math.random()) ) {
    case 0 -> g.setFill( Color.RED );
    case 1 -> g.setFill( Color.GREEN );
    case 2 -> g.setFill( Color.BLUE );
    case 3 -> g.setFill( Color.YELLOW );
}
```

I will choose the center points of the circles at random. Let's say that the width of the drawing area is given by a variable, `width`. Then we want a random value in the range `0` to `width-1` for the horizontal position of the center. Similarly, the vertical position of the center will be a random value in the range `0` to `height-1`. That leaves the size of the circle to be determined; I will make the radius of each circle equal to 50 pixels. We can draw the circle with a statement of the form `g.fillOval(x,y,w,h)`. However, in this command, `x` and `y` are not the coordinates of the center of the circle; they are the upper left corner of a rectangle drawn around the circle. To get values for `x` and `y`, we have to move back from the center of the circle by 50 pixels, an amount equal to the radius of the circle. The parameters `w` and `h` give the width and height of the rectangle, which have to be twice the radius, or 100 pixels in this case. Taking all this into account, here is a code segment for drawing a random circle:

```
centerX = (int)(width*Math.random());
centerY = (int)(height*Math.random());
g.fillOval( centerX - 50, centerY - 50, 100, 100 );
```

This code comes after the color-setting code given above. In the end, I found that the picture looks better if I also draw a black outline around each filled circle, so I added this code at the end:

```
g.setStroke( Color.BLACK );
g.strokeOval( centerX - 50, centerY - 50, 100, 100 );
```

Finally, to get a large number of circles, I put all of the above code into a `for` loop that runs for 500 executions. Here's a typical drawing from the program, shown at reduced size:

3.9.2 Drawing in a Program

Now, as you know, you can't just have a bunch of Java code standing by itself. The code has to be inside a subroutine definition that is itself inside a class definition. In fact, for my circle-drawing program, the complete subroutine for drawing the picture looks like this:

```
public void drawPicture(GraphicsContext g, int width, int
height) {

    g.setFill(Color.WHITE);
    g.fillRect(0, 0, width, height); // First, fill  with
a background color.

    // As an example, draw a large number of colored
disks.
    // To get a different picture, erase this code, and
substitute your own.

    int centerX;     // The x-coord of the center of a
disk.
    int centerY;     // The y-coord of the center of a
disk.
    int colorChoice; // Used to select a random color.
    int count;       // Loop control variable for counting
disks

    for (count = 0; count < 500; count++) {

        centerX = (int)(width*Math.random());
        centerY = (int)(height*Math.random());

        colorChoice = (int)(4*Math.random());
```

```
        switch (colorChoice) {
            case 0 -> g.setFill( Color.RED );
            case 1 -> g.setFill( Color.GREEN );
            case 2 -> g.setFill( Color.BLUE );
            case 3 -> g.setFill( Color.YELLOW );
        }

        g.fillOval( centerX - 50, centerY - 50, 100, 100
);
        g.setStroke(Color.BLACK);
        g.strokeOval( centerX - 50, centerY - 50, 100, 100
);
    }

} // end drawPicture()
```

This is the first subroutine definition that you have seen, other than `main()`, but you will learn all about defining subroutines in the next chapter. The first line of the definition makes available certain values that are used in the subroutine: the graphics context `g` and the `width` and `height` of the drawing area. These values come from outside the subroutine, but the subroutine can use them. The point here is that to draw something, you just have to fill in the inside of the subroutine, just as you write a program by filling in the inside of `main()`.

The subroutine definition still has to go inside a class that defines the program. In this case, the class is named *SimpleGraphicsStarter*, and the complete program is available in the sample source code file SimpleGraphicsStarter.java. You can run that program to see the drawing. You can use this sample program as a starting point for drawing your own pictures.

There's a lot in the program that you won't understand. To make your own drawing, all you have to do is erase the inside of the `drawPicture()` routine in the source code and substitute your own drawing code. You don't need to understand the rest.

(By the way, you might notice that the `main()` subroutine uses the word `static` in its definition, but `drawPicture()` does not. This has to do with the fact that `drawPicture` is a subroutine in an object rather than in a class. The difference between static and non-static subroutines is important but not

something that we need to worry about for the time being. It will become important for us in Chapter 5.)

3.9.3 Animation

We can extend the idea of drawing pictures to making **animations**. A computer animation is simply a sequence of individual pictures, displayed quickly one after the other. If the change from each picture to the next is small, the user will perceive the sequence of images as a continuous animation. Each picture in the animation is called a **frame**. The sample program SimpleAnimationStarter.java can be used as a starting point for writing animations. It contains a subroutine named `drawFrame()` that draws one frame in an animation. You can create an animation by filling in the definition of this subroutine. In addition to the graphics context and the width and height of the drawing area, you can use the value of two other variables in your code: `frameNumber` and `elapsedSeconds`. The `drawFrame` subroutine will automatically be called about 60 times per second. The variable `frameNumber` takes on the values 0, 1, 2, 3, ... in successive calls to the subroutine, and the value of `elapsedSeconds` is the number of seconds that the animation has been running. By using either of these variables in your code, you can draw a different picture each time `drawFrame()` is called, and the user will see the series of pictures as an animation.

As an example of animation, we look at drawing a set of nested rectangles. The rectangles will shrink towards the center of the drawing, giving an illusion of infinite motion. One frame from the animation looks like this:

Consider how to draw a picture like this one. The rectangles can be drawn with a `while` loop, which draws the rectangles starting from the one on the outside and moving in. Think about what variables will be needed and how they change from one iteration of the while loop to the next. Each time through the loop, the rectangle that is drawn is smaller than the previous one and is moved down and over a bit. The difference between two rectangles is in their sizes and in the coordinates of their upper left corners. We need variables to represent the width and height of the rectangle, which I call `rectWidth` and `rectHeight`. The x and y-coordinates of the upper left corner are the same, and they can be represented by the same variable. I call that variable `inset`, since it is the amount by which the edges of the rectangle are inset from the edges of the drawing area. The width and height decrease from one rectangle to the next, while the `inset` increases. The while loop ends when either the width or the height becomes less than or equal to zero. In general outline, the algorithm for drawing the rectangles in one frame is

```
Set the amount of inset for the first rectangle
Set the width and height for the first rectangle
Set the stroke color to black
while the width and height are both greater than zero:
    draw a rectangle (using the g.strokeRect subroutine)
    increase the inset (to move the next rectangle over
and down)
    decrease the width and height (to make the next
rectangle smaller)
```

In my program, 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** before drawing the next rectangle. The height also shrinks by 30 pixels each time through the loop.

The pseudocode is then easy to translate into Java, except that we need to know what initial values to use for the inset, width, and height of the first rectangle. To figure that out, we have to think about the fact that the picture is animated, so that what we draw will depend in some way on the frame number. From one frame to the next frame of the animation, the top-left corner of the outer rectangle moves over and down; that is, the `inset` for the outer rectangle increases from one frame to the next. We can make this happen by setting the inset for frame number 0 to 0, the inset for frame number 1 to 1, and so on. But that can't go on forever, or eventually all the rectangles would disappear. In fact, when the animation gets to frame 15, a new rectangle should appear at the outside of the drawing area—but it's not really a "new rectangle," it's just that the `inset` for the outer rectangle goes back to zero. So, as the animation proceeds, the inset should go through the sequence of values 0, 1, 2, ..., 14 over and over. We can accomplish that very easily by setting

```
inset = frameNumber % 15;
```

Finally, note that the first rectangle that is drawn in a frame fills the drawing area except for a border of size `inset` around the outside of the rectangle. This means that the width of the first rectangle is the width of the drawing area minus two times the inset, and similarly for the height. Here, then is the `drawFrame()` subroutine for the moving rectangle example:

```
public void drawFrame(GraphicsContext g, int frameNumber,
                              double elapsedSeconds, int
width, int height) {

    g.setFill(Color.WHITE);
    g.fillRect(0,0,width,height);  // Fill drawing area
with white.

    double inset; // Gap between edges of drawing area and
outer rectangle.
```

```
    double rectWidth, rectHeight;   // The size of one of
the rectangles.

    g.setStroke(Color.BLACK);  // Draw the rectangle
outlines in black.

    inset = frameNumber % 15 + 0.5;  // (The 0.5 is a
technicality that gives
                                     //  a sharper
picture.)

    rectWidth = width - 2*inset;
    rectHeight = height - 2*inset;

    while (rectWidth >= 0 && rectHeight >= 0) {
        g.strokeRect(inset, inset, rectWidth, rectHeight);
        inset += 15;       // rectangles are 15 pixels
apart
        rectWidth -= 30;
        rectHeight -= 30;
    }

}
```

You can find the full source code for the program is in the sample program MovingRects.java. Take a look! It's a neat effect. For another example of animation, see the sample program RandomCircles.java. That program adds one random colored disk to the picture in each frame; it illustrates the fact that the image from one frame is not automatically erased before the next frame is drawn.

Programming Exercises for Chapter 3

Exercise 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.

See the Solution

Exercise 3.2:

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.

See the Solution

Exercise 3.3:

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 consists 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.

See the Solution

Exercise 3.4:

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.

See the Solution

Exercise 3.5:

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 2.4.4. 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.

See the Solution

Exercise 3.6:

Exercise 3.2 asked you to find the number in the range 1 to 10000 that has the largest number of divisors. You only had to print out one such number. Revise the program so that it will print out **all** numbers that have the maximum number of divisors. Use an array as follows: As you count the divisors for each number, store each count in an array. Then at the end of the program, you can go through the array and print out all the numbers that have the maximum count. The output from the program should look something like this:

```
Among integers between 1 and 10000,
The maximum number of divisors was 64
Numbers with that many divisors include:
   7560
   9240
```

See the Solution

Exercise 3.7:

An example in Subsection 3.8.3 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 BirthdayProblem.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.)

See the Solution

Exercise 3.8:

Write a GUI program that draws a checkerboard. Base your solution on the sample program SimpleGraphicsStarter.java You will draw the checkerboard in the `drawPicture()` subroutine, after erasing the code that it already contains.

The checkerboard should be 400-by-400 pixels. You can change the size of the drawing area in `SimpleGraphicsStarter.java` by modifying the first two lines of the `start()` subroutine to set `width` and `height` to 400 instead of 800 and 600. A checkerboard contains 8 rows and 8 columns of squares. If the size of the drawing area is 400, that means that each square should be 50-by-50 pixels. The squares are red and black (or whatever other colors you choose). Here is a tricky way to determine whether a given square should be red or black: The rows and columns can be thought of as numbered from 0 to 7. If the row number of the square and the column number of the square 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 a reduced-size image of the checkerboard that you want to draw:

See the Solution

Exercise 3.9:

Often, some element of an animation repeats over and over, every so many frames. Sometimes, the repetition is "cyclic," meaning that at the end it jumps back to the start. Sometimes the repetition is "oscillating," like a back-and-forth motion where the second half is the same as the first half played in reverse.

Write an animation that demonstrates both cyclic and oscillating motions at various speeds. For cyclic motion, you can use a square that moves across the drawing area, then jumps back to the start, and then repeats the same motion over and over. For oscillating motion, you can do something similar, but the square should move back and forth between the two edges of the drawing area; that is, it moves left-to-right during the first half of the animation and then backwards from right-to-left during the second half. To write the program, you can start with a copy of the sample program SimpleAnimationStarter.java.

A cyclic motion has to repeat every N frames for some value of N. What you draw in some frame of the animation depends on the `frameNumber`. The `frameNumber` just keeps increasing forever. To implement cyclic motion, what you really want is a "cyclic frame number" that takes on the values 0, 1, 2, ..., (N-1), 0, 1, 2, ..., (N-1), 0, 1, 2, ... You can derive the value that you need from `frameNumber` simply by saying

```
cyclicFrameNumber = frameNumber % N;
```

Then, you just have to base what you draw on `cyclicFrameNumber` instead of on `frameNumber`. Similarly, for an oscillating animation, you need an "oscillation frame number" that takes on the values 0, 1, 2, ... (N-1), N, (N-1), (N-2), ... 2, 1, 0, 1, 2, and so on, repeating the back and forth motion forever. You can compute the value that you need with

```
oscilationFrameNumber = frameNumber % (2*N);
if (oscillationFrameNumber > N)
   oscillationFrameNumber = (2*N) -
oscillationFrameNumber;
```

Here is a screen shot from my version of the program. I use six squares. The top three do cyclic motion at various speeds, while the bottom three do oscillating motion. I drew black lines across the drawing area to separate the squares and to give them "channels" to move in.

See the Solution

Quiz on Chapter 3

Question 1:

What is an *algorithm*?

Question 2:

Explain briefly what is meant by "pseudocode" and how is it useful in the development of algorithms.

Question 3:

What is a *block statement?* How are block statements used in Java programs?

Question 4:

What is the main difference between a `while` loop and a `do..while` loop?

Question 5:

What does it mean to *prime* a loop?

Question 6:

Explain what is meant by an *animation* and how a computer displays an animation.

Question 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.

Question 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!

}
```

Question 9:

Write a code segment that will print out two *different* random integers selected from the range 1 to 10. All possible outputs should have the same probability. Hint: You can easily select two random numbers, but you have to account for the fact that the two numbers that you pick might be the same.

Question 10:

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.)

Question 11:

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);
    }
}
```

Question 12:

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);
   }
}
```

Question 13:

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;
}
```

Question 14:

Suppose that `numbers` is an array of type `int[]`. Write a code segment that will count and output the number of times that the number 42 occurs in the array.

Question 15:

Define the **range** of an array of numbers to be the maximum value in the array minus the minimum value. Suppose that `raceTimes` is an array of type `double[]`. Write a code segment that will find and print the range of `raceTimes`.

See the Answers

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.

Subroutines in Java can be either static or non-static. This chapter covers static subroutines. Non-static subroutines, which are used in true object-oriented programming, will be covered in the next chapter.

Contents of Chapter 4:

- Section 1: Black Boxes and Procedural Abstraction
- Section 2: Static Subroutines and Static Variables
- Section 3: Parameters
- Section 4: Return Values
- Section 5: Lambda Expressions
- Section 6: APIs, Packages, Modules, and Javadoc
- Section 7: More on Program Design
- Section 8: The Truth About Declarations

- Programming Exercises
- Quiz on This Chapter

Section 4.1

Black Boxes and Procedural Abstraction

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—on/off switch, 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, or 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 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 4.7, 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.

Subsection 3.1.4 introduced the idea of "control abstraction" to express the fact that a Java control structure lets the user work on a higher level than machine language, hiding the details of the process in the CPU that implements the control structure.

In a similar way, we say that a subroutine is a **procedural abstraction**. The interface of a subroutine is an abstraction that you can use to carry out some procedure without worrying about the details of how the procedure is actually implemented. This is just another, fancier way of saying that the subroutine can be used as a black box, but it brings us back to abstraction as a central concept in computer science.

An important aspect of abstraction in general is **information hiding**. An abstraction hides information about what is behind the abstraction, in its implementation. Information hiding by control abstractions is what makes it possible to use the same Java control structures on different types of CPU, with different underlying machine language implementations. Information hiding by procedural abstractions is what makes it possible for the inside of a subroutine to be changed without affecting the programs in which the subroutine is used. Again, this is just another term for the black box principle.

Section 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," as we will see later) helps control the confusion that might result from so many different names.

There is a basic distinction in Java between **static** and **non-static** subroutines. A class definition can contain the source code for both types of subroutine, but what's done with them when the program runs is very different. Static subroutines are easier to understand: In a running program, a static subroutine is a member of the class itself. Non-static subroutine definitions, on the other hand, are only there to be used when objects are created, and the subroutines themselves become members of the objects. Non-static subroutines only become relevant when you are working with objects. The distinction between static and non-static also applies to variables and to other things that can occur in class definitions. This chapter will deal with static subroutines and static variables almost exclusively. We'll turn to non-static stuff and to object-oriented programming in the next chapter.

A subroutine that is in a class or object 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, but I will continue to prefer the more general term "subroutine" in this chapter, at least for static subroutines.

However, you should start thinking of the terms "method" and "subroutine" as being essentially synonymous as far as Java is concerned. Other terms that you might see used to refer to subroutines are "procedures" and "functions." (I generally use the term "function" only for subroutines that compute and return a value, but in some programming languages, it is used to refer to subroutines in general.)

4.2.1 Subroutine Definitions

A subroutine must be defined somewhere. The definition has to include the name of the subroutine, enough information to make it possible to call the subroutine, and the code that will be executed each time the subroutine is called. 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 programs in Section 3.9. 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. It can be a type name such as *String* or int or even an array type such as `double[]`. 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. If a 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" is 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. (You will learn more about packages later in this chapter, in Section 4.6.) 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`". In this case, the type for the parameter is the array type `String[]`.

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 objects 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 specified 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;
    System.out.println();
    System.out.print("What is your first guess? ");
    while (true) {
       usersGuess = TextIO.getInt();  // Get the user's
guess.
       guessCount++;
       if (usersGuess == computersNumber) {
          System.out.println("You got it in " + guessCount
                  + " guesses!  My number was " +
computersNumber);
          break;  // The game is over; the user has won.
       }
       if (guessCount == 6) {
          System.out.println("You didn't get the number in
6 guesses.");
          System.out.println("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)
          System.out.print("That's too low.  Try again:
");
       else if (usersGuess > computersNumber)
          System.out.print("That's too high.  Try again:
");
    }
    System.out.println();
} // 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()`; it will **not** contain the definition of `playGame`. You can put the definition of `playGame()` either before or after the `main()` routine. Java is

not very picky about having the subroutine definitions in 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).

```
import textio.TextIO;

public class GuessingGame {

   public static void main(String[] args) {
      System.out.println("Let's play a game.  I'll pick a
number between");
      System.out.println("1 and 100, and you try to guess
it.");
      boolean playAgain;
      do {
         playGame();  // call subroutine to play one game
         System.out.print("Would you like to play again?
");
         playAgain = TextIO.getlnBoolean();
      } while (playAgain);
      System.out.println("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;
       System.out.println();
       System.out.print("What is your first guess? ");
       while (true) {
          usersGuess = TextIO.getInt();  // Get the user's
guess.
          guessCount++;
          if (usersGuess == computersNumber) {
```

```
              System.out.println("You got it in " +
guessCount
                      + " guesses!  My number was " +
computersNumber);
              break;  // The game is over; the user has
won.
           }
           if (guessCount == 6) {
              System.out.println("You didn't get the number
in 6 guesses.");
              System.out.println("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)
              System.out.print("That's too low.  Try again:
");
           else if (usersGuess > computersNumber)
              System.out.print("That's too high.  Try
again: ");
        }
        System.out.println();
    } // 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. Another term for them is **global variable**.

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 as a whole, 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 static member variable in the *Math* class. If `numberOfPlayers` is a public

static member variable in a class named `Poker`, then code in the `Poker` class would refer to it simply as `numberOfPlayers`, while code in another class would refer to it as `Poker.numberOfPlayers`.

As an example, let's add a couple of static member variables to the `GuessingGame` class that we wrote earlier in this section. We add a variable named `gamesPlayed` to keep track of how many games the user has played and another variable named `gamesWon` to keep track of the number of games that the user has won. The variables are declared as static member variables:

```
static int gamesPlayed;
static int gamesWon;
```

In the `playGame()` routine, we always add 1 to `gamesPlayed`, and we add 1 to `gamesWon` if the user wins the game. At the end of the `main()` routine, we print out the values of both variables. It would be impossible to do the same thing with local variables, since both subroutines need to access the variables, and local variables exist in only one subroutine. Furthermore, global variables keep their values between one subroutine call and the next. Local variables do not; a local variable gets a new value each time that the subroutine that contains it is called.

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. The default values are the same as those that are used when initializing the elements of an array: For numeric variables, the default value is zero; for boolean variables, the default is `false`; for char variables, it's the character that has Unicode code number zero; and for objects, such as *Strings*, the default initial value is the special value `null`.

Since they are of type int, the static member variables `gamesPlayed` and `gamesWon` automatically get zero as their initial value. This happens to be the correct initial value for a variable that is being used as a counter. You can, of course, assign a value to a variable at the beginning of the `main()` routine if you are not satisfied with the default initial value, or if you want to make the initial value more explicit.

Here's the revised version of `GuessingGame.java`. The changes from the above version are shown in red italic:

```
import textio.TextIO;

public class GuessingGame2 {

    static int gamesPlayed;   // The number of games
played.
    static int gamesWon;      // The number of games won.

    public static void main(String[] args) {
       gamesPlayed = 0;
       gamesWon = 0;  // This is actually redundant, since
0 is
                      //                 the default
initial value.
       System.out.println("Let's play a game.  I'll pick a
number between");
       System.out.println("1 and 100, and you try to guess
it.");
       boolean playAgain;
       do {
          playGame();  // call subroutine to play one game
          System.out.print("Would you like to play again?
");
          playAgain = TextIO.getlnBoolean();
       } while (playAgain);
       System.out.println();
       System.out.println("You played " + gamesPlayed + "
games,");
       System.out.println("and you won " + gamesWon + " of
those games.");
       System.out.println("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.
        gamesPlayed++;  // Count this game.
        computersNumber = (int)(100 * Math.random()) + 1;
```

```
                 // The value assigned to computersNumber
is a randomly
                 //    chosen integer between 1 and 100,
inclusive.
        guessCount = 0;
        System.out.println();
        System.out.print("What is your first guess? ");
        while (true) {
           usersGuess = TextIO.getInt();  // Get the
user's guess.
           guessCount++;
           if (usersGuess == computersNumber) {
              System.out.println("You got it in " +
guessCount
                     + " guesses!  My number was " +
computersNumber);
              gamesWon++;  // Count this win.
              break;       // The game is over; the user
has won.
           }
           if (guessCount == 6) {
              System.out.println("You didn't get the
number in 6 guesses.");
              System.out.println("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)
              System.out.print("That's too low.  Try
again: ");
           else if (usersGuess > computersNumber)
              System.out.print("That's too high.  Try
again: ");
        }
        System.out.println();
    } // end of playGame()

} // end of class GuessingGame2
```

(By the way, notice that in my example programs, I didn't mark the static subroutines or variables as being `public` or `private`. You might wonder

what it means to leave out both modifiers. Recall that global variables and subroutines with no access modifier can be used anywhere in the same package as the class where they are defined, but not in other packages. Classes that don't declare a package are in the default package. So, any class in the default package would have access to `gamesPlayed`, `gamesWon`, and `playGame()`—and that includes most of the classes in this book. In fact, it is considered to be good practice to make member variables and subroutines `private`, unless there is a reason for doing otherwise. (But then again, it's also considered good practice to avoid using the default package.))

Section 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 can 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. But notice that there is nothing in the subroutine definition that gives a value to the parameter! The parameter gets its initial value from **outside** the subroutine. When the subroutine is called, a value must be provided for the parameter in the subroutine call statement. This value will be assigned to `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 the subroutine can be called by saying "`print3NSequence(K);`". When the computer executes this subroutine call statement, it takes the value of the variable `K`, assigns that value to `startingValue`, and then executes 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, *String*, or `double[]`. 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 ints can legally be assigned to doubles. 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.** By the time the computer starts executing the statements in the subroutine, the formal parameters have **already** been assigned initial values! The computer automatically assigns values to the formal parameters before it starts executing the code inside the subroutine. The values come from the actual parameters in the subroutine call statement. Remember that a subroutine is not independent. It is called by some other routine, and it is the subroutine call statement'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. 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(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('A')` calls the subroutine with signature `println(char)`. 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 char, which is different from printing out a boolean, and so forth—so that each of these operations requires a different subroutine.

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 subroutines defined as:

```
int    getln() { ... }
double getln() { ... }
```

This is why in the *TextIO* class, the subroutines for reading different types are not all named `getln()`. In a given class, there can only be one routine that has the name `getln` with no parameters. So, the input routines in *TextIO*

are distinguished by having different names, such as `getlnInt()` and `getlnDouble()`.

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 4.7.

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 implies that there is one parameter, of type int, that represents the "given integer" whose divisors are to be printed. 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 )  // Does 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 `private` 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, and we are told that the subroutine is `private`, 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.
 */
private 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 defined above 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 that contains 25 copies of character number `i` from the string. So, we get:

```
/**
 * For each character in str, write a line of output
 * containing 25 copies of that character.
 */
private 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 then use `printRowsFromString` in a `main()` routine such as

```
public static void main(String[] args) {
    String inputLine;  // Line of text input by user.
    System.out.print("Enter a line of text: ");
    inputLine = TextIO.getln();
    System.out.println();
    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 Array Parameters

It's possible for the type of a parameter to be an array type. This means that an entire array of values can be passed to the subroutine as a single parameter. For example, we might want a subroutine to print all the values in an integer array in a neat format, separated by commas and enclosed in a pair of square brackets. To tell it which array to print, the subroutine would have a parameter of type `int[]`:

```
static void printValuesInList( int[] list ) {
    System.out.print('[');
    int i;
    for ( i = 0; i < list.length; i++ ) {
        if ( i > 0 )
            System.out.print(','); // No comma in front of
list[0]
        System.out.print(list[i]);
    }
    System.out.println(']');
}
```

To use this subroutine, you need an actual array. Here is a legal, though not very realistic, code segment that creates an array just to pass it as an argument to the subroutine:

```
int[] numbers;
numbers = new int[3];
numbers[0] = 42;
numbers[1] = 17;
numbers[2] = 256;
printValuesInList( numbers );
```

The output produced by the last statement would be `[42,17,256]`.

4.3.6 Command-line Arguments

The `main` routine of a program has a parameter of type `String[]`. When the main routine is called, some actual array of *String* must be passed to `main()` as the value of the parameter. The system provides the actual parameter when it calls `main()`, so the values come from outside the program. Where do the strings in the array come from, and what do they mean? The strings in the array are **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. The system takes the command-line arguments, puts them into an array of strings, and passes that array to `main()`.

For example, if the name of the program 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:");
         int i;
         for ( i = 0; i < args.length; i++ )
            System.out.println("   " + args[i]);
      }
   } // end main()

} // end class CLDemo
```

Note that the parameter, `args`, can be an array of length zero. This just means that the user did not include any command-line arguments when running the program.

In practice, command-line arguments are often used to pass the names of files to a program. For example, consider the following program for making a copy of a text file. It does this by copying one line at a time from the original file to the copy, using TextIO. The function `TextIO.eof()` is a boolean-valued function that is `true` if the end of the input file has been reached.

```
input textio.TextIO;

/**
 *  Requires two command line arguments, which must be
file names.  The
 *  first must be the name of an existing file.  The
second is the name
 *  of a file to be created by the program.  The contents
of the first file
 *  are copied into the second.  WARNING:  If the second
file already
 *  exists when the program is run, its previous contents
will be lost!
 *  This program only works for plain text files.
 */
public class CopyTextFile {

    public static void main( String[] args ) {
        if (args.length < 2 ) {
            System.out.println("Two command-line arguments
are required!");
            System.exit(1);
        }
        TextIO.readFile( args[0] );   // Open the original
file for reading.
        TextIO.writeFile( args[1] );  // Open the copy
file for writing.
        int lineCount;  // Number of lines copied
        lineCount = 0;
        while ( TextIO.eof() == false ) {
            // Read one line from the original file and
write it to the copy.
            String line;
            line = TextIO.getln();
            TextIO.putln(line);
```

```
            lineCount++;
        }
        System.out.printf( "%d lines copied from %s to
%s%n",
                                lineCount, args[0],
args[1] );
    }

}
```

Since most programs are run in a GUI environment these days, command-line arguments aren't as important as they used to be. But at least they provide a nice example of how array parameters can be used.

4.3.7 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 the 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 do the same 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. For this to work, the subroutine call would have to be in the "try" part of the statement.

4.3.8 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.

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 arrays and 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. Such a variable is said to be **global** to the subroutine, as opposed to the local variables defined inside the subroutine. A global variable can be used in the entire class in which it is defined and, if it is not `private`, in other classes as well. 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 values of the global variables, `gamesPlayed` and `gamesWon`, are computed inside a subroutine and are 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.

Section 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 an expression that could legally be assigned to a variable whose type is specified by the return type of the function.) 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, a function must definitely return some value (or throw an exception), 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.

Note that a `return` inside a loop will end the loop as well as the subroutine that contains it. Similarly, a `return` in a `switch` statement breaks out of the `switch` statement as well as the subroutine. So, you will sometimes use `return` in contexts where you are used to seeing a `break`.

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 the 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:

```
/**
 * This 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) + 0.001);
        // 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.  I added
        // the 0.001 because computations with double
values are not
        // exact, and I worry that, for example,
Math.sqrt(49) might
        // be computed as 6.999... instead of as 7.0.)

    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.

```
import textio.TextIO;

/**
 * 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) {

      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;   // Starting point for sequence, specified
by the user.
      do {
         System.out.println("Enter a starting value;");
         System.out.print("To end the program, enter 0:
");
         K = TextIO.getlnInt();  // 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.
```

```
      System.out.println("The 3N+1 sequence starting from
" + N);
      System.out.println();
      System.out.printf("%8d", N);  // 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
             System.out.println();  // ...then output a
carriage return
             onLine = 0;      // ...and note that there
are no terms
                              //               on the new
line.
          }
          System.out.printf("%8d", N);  // Print this term
in an 8-char column.
          onLine++;   // Add 1 to the number of terms on
this line.
      }

      System.out.println();  // end current line of output
      System.out.println();  // and then add a blank line
      System.out.println("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.

Section 4.5

Lambda Expressions

IN A RUNNING PROGRAM, A SUBROUTINE IS JUST a bunch of binary numbers (representing instructions) stored somewhere in the computer's memory. Considered as a long string of zeros and ones, a subroutine doesn't seem all that different from a data value such as, for example, as an integer, a string, or an array, which is also represented as a string of zeros and ones in memory. We are used to thinking of subroutines and data as very different things, but inside the computer, a subroutine is just another kind of data. Some programming languages make it possible to work with a subroutine as a kind of data value. In Java 8, that ability was added to Java in the form of something called **lambda expressions.**

Lambda expressions are becoming more and more common in Java programs. They are especially useful for working with the stream API that will be covered in Section 10.6 and with the JavaFX GUI toolkit. It will definitely be useful to know about them before we cover GUI programming in Chapter 6. However, we won't encounter them again until near the end of Chapter 5, so you can skip this section for now if you want.

4.5.1 First-class Functions

Lambda is a letter in the Greek alphabet that was used by the mathematician Alonzo Church in his study of computable functions. His lambda notation makes it possible to define a function without giving it a name. For example, you might think that the notation x^2 is a perfectly good way of representing a function that squares a number, but in fact, it's an expression that represents the result of squaring x, which leaves open the question of what x represents. We can define a function with x as a dummy parameter:

```
static double square( double x ) {
    return x*x;
```

```
}
```

but to do that, we had to name the function *square*, and that name becomes a permanent part of the program—which is overkill if we just want to use the function once. Alonzo Church introduced the notation $lambda(x).x^2$ to represent "the function of x that is given by x^2" (except using the Greek letter instead of the word "lambda"). This notation is a kind of function literal that represents a value of type "function" in the same way that 42 is an integer literal that represents a value of type int.

Having function literals is the starting point for thinking of a function as just another kind of data value. Once we do that, we should be able to do the same things with functions that we can do with other values, such as assign a function to a variable, pass a function as a parameter to a subroutine, return a function as the value of subroutine, or even make an array of functions. A programming language that allows you to do all those things with functions is said to have "first-class functions" or "functions as first-class objects."

In fact, you can do all of those things with Java lambda expressions. Java's notation is different from the one used by Alonzo Church, and in spite of the name "lambda expression" it does not even use the word lambda. In Java, the lambda expression for a squaring function like the one above can be written

```
x  ->  x*x
```

The operator `->` is what makes this a lambda expression. The dummy parameter for the function is on the left of the operator, and the expression that computes the value of the function is on the right. You might see an expression like this one being passed as an actual parameter to a subroutine, assigned to a variable, or returned by a function.

So are functions now first-class in Java? I'm not quite sure. There are some cool things that can be done in other languages but can't be done in Java. For example, in Java we can assign the above expression to a variable named, say, *sqr*, but we can't then use *sqr* as if it actually is a function. For example, we can't say *sqr*(42). The problem, really, is that Java is a strongly typed language; to have a variable named *sqr*, we must declare that variable and give it a type. But what sort of type would be appropriate for a value that is a

function? The answer in Java is something called a **functional interface**, which we turn to next.

But first one more note: Lambda expressions in Java can actually represent arbitrary subroutines, not just functions. Nevertheless, it is the term "function" that is usually associated with them, rather than "subroutine" or "method."

4.5.2 Functional Interfaces

To know how a subroutine can be legally used, you need to know its name, how many parameters it requires, their types, and the return type of the subroutine. A functional interface specifies this information about one subroutine. A functional interface is similar to a class, and it can be defined in a .java file, just like a class. However, its content is just a specification for a single subroutine. Here is an example:

```
public interface FunctionR2R {
    double valueAt( double x );
}
```

This code would be in a file named *FunctionR2R.java*. It specifies a function named *valueAt* with one parameter of type double and a return type of double. (The **name** of the parameter, *x*, is not really part of the specification, and it's a little annoying that it has to be there.) Here is another example:

```
public interface ArrayProcessor {
    void process( String[] array, int count );
}
```

Java comes with many standard functional interfaces. One of the most important is a very simple one named *Runnable*, which is already defined in Java as

```
public interface Runnable {
    public void run();
}
```

I will use these three functional interfaces for examples in this section.

"Interfaces" in Java can be much more complicated than functional interfaces. You will learn more about them in Section 5.7. But it is only functional interfaces that are relevant to lambda expressions: a functional interface provides a template for a subroutine that might be represented by a lambda expression. The name of a functional interface is a **type**, just as *String* and double are types. That is, it can be used to declare variables and parameters and to specify the return type of a function. When a type is a functional interface, a value for that type can be given as a lambda expression.

4.5.3 Lambda Expressions

A lambda expression represents an anonymous subroutine, that is, one without a name. But it does have a formal parameter list and a definition. The full syntax is:

```
( parameter-list )  ->  { statements }
```

As with a regular subroutine, the **parameter-list** can be empty, or it can be a list of parameter declarations, separated by commas, where each declaration consists of a type followed by a parameter name. However, the syntax can often be simplified. First of all, the parameter types can be omitted, as long as they can be deduced from the context. For example, if the lambda expression is known to be of type *FunctionR2R*, then the parameter type must be double, so it is unnecessary to specify the parameter type in the lambda expression. Next, if there is exactly one parameter and if its type is not specified, then the parentheses around the parameter list can be omitted. On the right-hand side of the `->`, if the only thing between the braces, `{` and `}`, is a single subroutine call statement, then the braces can be omitted. And if the right-hand side has the form `{ return` **expression**; `}`, then you can omit everything except the **expression**.

For example, suppose that we want a lambda expression to represent a function that computes the square of a double value. The type of such a function can be the *FunctionR2R* interface given above. If *sqr* is a variable of type *FunctionR2R*, then the value of the function can be a lambda expression, which can be written in any of the following forms:

```
sqr = (double x) -> { return x*x; }; // The full lambda
expression syntax!
sqr = (x) -> { return x*x; };
sqr = x -> { return x*x; };
sqr = x -> x*x;
sqr = (double fred) -> fred*fred;
sqr = (z) -> z*z;
```

The last two statements are there to emphasize that the parameters in a lambda expression are dummy parameters; their names are irrelevant. The six lambda expressions in these statements all define exactly the same function. Note that the parameter type double can be omitted because the compiler knows that `sqr` is of type *FunctionR2R*, and a *FunctionR2R* requires a parameter of type double. A lambda expression can only be used in a context where the compiler can deduce its type, and the parameter type has to be included only in a case where leaving it out would make the type of the lambda expression ambiguous.

Now, in Java, the variable *sqr* as defined here is not quite a function. It is a value of type *FunctionR2R*, which means that it **contains** a function named *valueAt*, as specified in the definition of interface *FunctionR2R*. The full name of that function is *sqr.valueAt*, and we must use that name to call the function. For example: `sqr.valueAt`(42) or `sqr.valueAt(x) + sqr.valueAt(y)`.

When a lambda expression has two parameters, the parentheses are not optional. Here is an example of using the *ArrayProcessor* interface, which also demonstrates a lambda expression with a multiline definition:

```
ArrayProcessor concat;
concat = (A,n) -> { // parentheses around (A,n) are
required!
    String str;
    str = "";
    for (int i = 0; i < n; i++)
        str += A[i];
    System.out.println(str);
};  // The semicolon marks the end of the assignment
statement;
    //      it is not part of the lambda expression.

String[] nums;
```

```
nums = new String[4];
nums[0] = "One";
nums[1] = "Two";
nums[2] = "Three";
nums[3] = "Four";
for (int i = 1; i < nums.length; i++) {
   concat.process( nums, i );
}
```

This will print out

```
One
OneTwo
OneTwoThree
OneTwoThreeFour
```

Things get more interesting when a lambda expression is used as an actual parameter, which is the most common use in practice. For example, suppose that the following function is defined:

```
/**
 *  For a function f, compute f(start) + f(start+1) + ...
+ f(end).
 *  The value of end should be >= the value of start.
 */
static double sum( FunctionR2R f, int start, int end ) {
    double total = 0;
    for (int n = start; n <= end; n++) {
        total = total + f.valueAt( n );
    }
    return total;
}
```

Note that since *f* is a value of type *FunctionR2R*, the value of *f* at *n* is actually written as *f.valueAt*(*n*). When the function *sum* is called, the first parameter can be given as a lambda expression that matches the type *FunctionR2R*. For example:

```
System.out.print("The sum of n squared for n from 1 to 100
is ");
System.out.println( sum( x -> x*x, 1, 100 ) );
System.out.print("The sum of 2 raised to the power n, for
n from 1 to 10 is ");
System.out.println( sum( num -> Math.pow(2,num), 1, 10 )
);
```

As another example, suppose that we have a subroutine that performs a given task several times. The task can be specified as a value of type *Runnable*:

```
static void doSeveralTimes( Runnable task, int repCount )
{
    for (int i = 0; i < repCount; i++) {
        task.run();  // Perform the task!
    }
}
```

We could then say "Hello World" ten times by calling

```
doSeveralTimes( () -> System.out.println("Hello World"),
10 );
```

Note that for a lambda expression of type *Runnable*, the parameter list is given as an empty pair of parentheses. Here is an example in which the syntax is getting rather complicated:

```
doSeveralTimes( () -> {
        // count from 1 up to some random number between 5
and 25
    int count = 5 + (int)(21*Math.random());
    for (int i = 1; i <= count; i++) {
        System.out.print(i + " ");
    }
    System.out.println();
}, 100);
```

This is a single subroutine call statement in which the first parameter is a lambda expression that extends over multiple lines. The second parameter is 100, and the semicolon on the last line ends the subroutine call statement.

We have seen examples of assigning a lambda expression to a variable and of using one as an actual parameter. Here is an example in which a lambda expression is the return value of a function:

```
static FunctionR2R makePowerFunction( int n ) {
   return x -> Math.pow(x,n);
}
```

Then *makePowerFunction*(2) returns a *FunctionR2R* that computes the square of its parameter, while *makePowerFunction*(10) returns a

FunctionR2R that computes the 10-th power of its parameter. This example also illustrates the fact that a lambda expression can use other variables in addition to its parameter, such as *n* in this case (although there are some restrictions on when that can be done).

4.5.4 Method References

Suppose that we want a lambda expression to represent the square root function as a value of type *FunctionR2R*. We could write it as `x -> Math.sqrt(x)`. However, this lambda expression is a simple wrapper for a *Math.sqrt* function that already exists. Instead of writing out the lambda expression, that function can be written as a **method reference**, which takes the form `Math::sqrt`. (Recall that in Java, "method" is another word for "subroutine.") This method reference is just a shorthand for the lambda expression, and it can be used wherever that lambda expression could be used, such as in the *sum* function defined above:

```
System.out.print("The sum of the square root of n for n
from 1 to 100 is ");
System.out.println( sum( Math::sqrt, 1, 100 ) );
```

It would be nice if we could simply use the name *Math.sqrt* here instead of introducing a new notation with `::`, but the notation *Math.sqrt* was already defined to mean a **variable** named *sqrt* in the *Math* class.

More generally, a lambda expression that simply calls an existing static method can be written as a method reference of the form

```
classname :: method-name
```

Furthermore, this notation extends to methods that are in objects rather than classes. For example, if *str* is a *String*, then *str* contains the method *str.length*(). The method reference `str::length` could be used as a lambda expression of type *SupplyInt*, where *SupplyInt* is the functional interface

```
public interface SupplyInt {
    int get( );
}
```

Section 4.6

APIs, Packages, Modules, 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.

4.6.1 Toolboxes

Someone who wanted to program for the original Macintosh computers—and to produce programs that look and behave the way users expected them to—had to deal with the "Macintosh Toolbox," a collection of well over a thousand different subroutines. There were 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 were other routines for creating menus and for reacting to user selections from menus. Aside from the user interface, there were 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 **Application Programming Interface**, associated with the toolbox. The API is the abstraction through which you access the functionality of the software in 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 part of 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 MacOS, 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** of some parts of the API 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 is the power of abstraction. It means that you only need to learn one API to program for a wide variety of platforms.

4.6.2 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 2.6.5. 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 graphical user interface classes. Another package, "`javax`", contains the classes used by the Swing graphical user interface, as well as many other classes. And "`javafx`," which is not a standard part of Java, contains the JavaFX API that is used for GUI programming in this textbook.

A package can contain both classes and other packages. A package that is contained in another package is sometimes called a "subpackage." The `java` and `javafx` packages both contain subpackages. One of the subpackages of `java`, for example, is named "`util`". Since `util` is contained within `java`, its full name is actually `java.util`. This package contains a variety of utility classes, including the *Scanner* class that was discussed in Subsection 2.4.6.

The `java` package includes several other subpackages, such as `java.io`, which provides facilities for input/output, and `java.net`, which deals with network communication. 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 subpackages, the classes in those subpackages, 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:

Subroutines nested in classes nested in two layers of packages.
The full name of sqrt() is java.lang.Math.sqrt().

Similarly, the package `javafx` contains a package `javafx.scene`, which in turn contains `javafx.scene.control`. This package contains classes that represent GUI components such as buttons and input boxes. Another subpackage, `javafx.scene.paint`, contains class *Color* and other classes that define ways to fill and stroke a shape.

The standard Java API includes thousands of classes in hundreds of packages. Many of the classes 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 basic API for Java 8 can be found at

https://docs.oracle.com/javase/8/docs/api/

and for JavaFX 8 at

https://docs.oracle.com/javase/8/javafx/api/toc.htm

See the subsection about "modules," below, for a discussion of changes that were made the language after Java 8 and for links to the documentation for Java 17. However, for the purposes of this textbook, you will probably find that the Java 8 documentation is easier to use, and the information that it provides is still relevant.

Even an expert programmer won't be familiar with the entire Java 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.

4.6.3 Using Classes from Packages

Let's say that you want to use the class `javafx.scene.paint.Color` in a program that you are writing. This is the full name of class *Color* in package `javafx.scene.paint`. Like any class, `javafx.scene.paint.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 *Color*. You could say:

```
javafx.scene.paint.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, and you will hardly ever see full names like this used in a program. Java makes it possible to avoid using the full name of a class by **importing** the class. If you put

```
import javafx.scene.paint.Color;
```

at the beginning of a Java source code file, then, in the rest of the file, you can abbreviate the full name `javafx.scene.paint.Color` to just the simple

name of the class, which is `Color`. Note that the `import` line comes at the start of a file (after the `package` statement, if there is one) 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 javafx.scene.paint.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. For example, you can import all the classes from `java.util` by saying

```
import java.util.*;
```

The "`*`" is a **wildcard** that matches every class in the package. (However, it does not match subpackages; for example, you **cannot** import the entire contents of all the subpackages 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.

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 a class 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. You can still use both classes in your program: 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 files that defines those classes must begin with the line

```
package utilities;
```

This would come even before any `import` directive in that file. Furthermore, the source code file would be placed in a folder with the same name as the package, "utilities" in this example. And a class that is in a subpackage must be in a subfolder. For example, a class in a package named `utilities.net` would be in folder named "net" inside a folder named "utilities". 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 classes from 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 such as class *TextIO* in package `textio`, the classes written for this book are not in packages. For the

purposes of this book, you need to know about packages mainly so that you will be able to import *TextIO* and classes from the standard packages. The standard 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 they will be part of the installed JDK.

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.

4.6.4 About Modules

Starting with Java 9, a major change was made to the large-scale structure of Java with the introduction of **modules**. A module is a collection of packages, so it represents yet another level of containment: Modules contain packages which contain classes which contain variables and methods. A package does not have to be in a module to be used, but all of the standard classes in Java and in JavaFX have been divided into a set of modules.

Modules were introduced for several reasons. A major reason is to provide better access control. Before modules, a class that is declared `public` can be used anywhere, from any class in any package, as can its public variables and methods. For a class that is defined in a module, on the other hand, "public" only automatically means public within the module where it is defined. However, a module can explicitly **export** a package. Exporting a package from a module makes the public classes in the package accessible from anywhere, including from other modules and from classes that are not part of any module. (It is even possible to export a package just to certain specified modules, providing an even finer level of access control.) The upshot is that it is now possible to have entire packages that are essentially private: They provide services to other packages in the same module, but are invisible from outside that module. So a module is another kind of black box, and a non-exported package is part of its hidden implementation. Of course,

modularity on this scale is really only important for very large-scale applications.

Another motivation for modules is the sheer size of the standard JRE (Java Runtime Environment), which includes all of the standard classes. A given application will use only a small part of the standard runtime. Modularization makes it possible to construct smaller, custom JREs that contain only the modules that are required by an application. The JDK includes a *jlink* command for making custom runtimes, which can include modules that define an application as well as the standard modules that are required to run that application. That runtime can then be distributed as a standalone application that can be executed even by people who have not installed a JDK on their computer. But just as for the JDK itself, different versions of the custom runtime will be needed for Windows, for MacOS, and for Linux. Furthermore, when security updates are made to the JDK, they are not automatically applied to custom runtimes, so the application developer takes on the responsibility of updating custom runtimes. Once again, this is really only useful for fairly large applications.

In a JDK for Java 9 or later, compiled class files from the standard modules are stored together in a file named *modules* inside a directory named *lib* in the main JDK directory. This is a so-called "jimage file," and there is a command-line tool named *jimage* for working with such files. If you use the *jlink* tool to create a custom runtime, part of what it does is to create a custom *modules* file containing just the modules that are needed by the runtime. In the JDK 17 on my Linux computer, *modules* is a 127 megabyte file containing 26401 classes in 835 packages in 70 modules. The JDK directory also has a subdirectory named *jmods* that contains the modules in another form. However, it is not required for compiling and running programs and, as far as I can tell, is meant mostly for use by *jlink*.

Modules in the JDK include, for example, *java.base* (which contains the basic modules such as `java.lang` and `java.util`) and `java.desktop` (which include packages for the Swing GUI toolkit). JavaFX packages include *javafx.base*, *javafx.control*, *javafx.graphics*, and a few that are less generally useful. The API documentation for modular versions of Java is divided into modules, then into packages, and finally into classes. This

makes the documentation harder to browse than in older versions of Java. However, the documentation web site does have an effective search feature. As I write this, the documentation for Java 17 and for JavaFX 17 is available at:

https://docs.oracle.com/en/java/javase/17/docs/api/index.html

https://openjfx.io/javadoc/17/

A class can be defined outside of any module, and it is possible for that class to use packages from modules, provided that those packages are exported by the modules where they are defined. In particular, a programmer can use classes from the JDK without ever thinking about modules or knowing that they exist. This applies to all the command-line programs in this book. However, when using Java 9 or later, things are different for GUI programs that use JavaFX, which has been removed from the JDK and is distributed as a separate set of modules. As we saw in Section 2.6, when you compile or run a JavaFX program, you need to specify a module path that includes the JavaFX modules, and you need to provide an `--add-modules` option. (In Section 2.6, the value for `--add-modules` was given as `ALL-MODULE-PATH`, which lets the program access any modules that are found on the module path. An alternative is to specify a list of names of just those modules that are actually used by the program.)

Aside from using modules with JavaFX and the basic background information in this section, this textbook does not cover modules.

4.6.5 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 many of the examples in this book.

Note that a 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. The book will not teach you HTML, 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 `&` and `<`.)

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 four tags: `@author`, `@param`, `@return`, and `@throws`. The `@author` tag can be used only for a class, and should be followed by the name of the author. The other three tags are used in Javadoc comments for a subroutine 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 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 the `javadoc` tool in a command line interface similarly to the way that the `javac` and `java` commands are used. Javadoc can also be applied in the integrated development environments that were also discussed in Section 2.6. I won't go into any of the details here; consult the documentation for your programming environment.

4.6.6 Static Import

Before ending this section, I will mention an extension of the `import` directive. We have seen that `import` makes it possible to refer to a class such as `java.util.Scanner` using its simple name, *Scanner*. 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`.

There is another 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. That 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 say `out.println` instead of `System.out.println`. If you are going to work extensively with the *Math* class, you might 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. And you could import the *getlnInt* function from *TextIO* using

```
import static textio.TextIO.getlnInt;
```

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.

Section 4.7

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.

4.7.1 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, because you haven't lived up to your side of the deal.

A postcondition of a subroutine represents the other side of the contract. It represents an obligation of the subroutine. 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. Or, if it only makes

sense to call the subroutine at certain times, the precondition might refer to the state that the program must be in when the subroutine is called.

The postcondition of a subroutine, on the other hand, 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.)

4.7.2 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. The API that I will use here is defined in two classes that I have written: Mosaic.java, which in turn depends on MosaicCanvas.java. To compile and run a program that uses the API, the classes *Mosaic* and *MosaicCanvas* must be available. That is, the files `Mosaic.java` and `MosaicCanvas.java`, or the corresponding compiled class files, must be in the same folder as the class that defines the program. (You can download them from this textbooks's web site.)

So, suppose that I have access to 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. (Remember that a Javadoc comment comes before the thing that it is commenting on.)

```
/**
 * Opens a "mosaic" window on the screen.  This subroutine
should be called
 * before any of the other Mosaic subroutines are used.
The program will end
 * when the user closes the window.
 *
 * Precondition:   The parameters rows, cols, h, and w 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)


/**
 * 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 static 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.open` rather than simply `open`.

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, and it would be good to document it with a `@throws` doc tag in the Javadoc comment. 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 second call will simply be ignored.) 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 16 rows and 20 columns of squares in my mosaic, so setting the current position to be in the center means setting `currentRow` to 8 and `currentColumn` to 10. I already have a subroutine, `Mosaic.open()`, to open the window. 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(16,20,25,25)
fillWithRandomColors();
currentRow = 8;       // Middle row, halfway down the
window.
currentColumn = 10;   // Middle column.
while ( true ) { // Program ends when user closes the
window.
    changeToRandomColor(currentRow, currentColumn);
    randomMove();
}
```

With the proper wrapper, this is essentially the `main()` routine of my program. It turns out I decided to make one small modification after running the completed program: To prevent the animation from running too fast, I added the line "`Mosaic.delay(10);`" 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() {
   int row, column;
   for ( row = 0; row < 16; row++ )
      for ( 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 a precondition of several of the *Mosaic* subroutines). Rather than let this happen, I decide to move the disturbance to the opposite edge of the grid by setting `currentRow` to 15. (Remember that the 16 rows are numbered from 0 to 15.) 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 = 15;   // move it to the opposite
edge.
   }
   case 1 -> {  // move right
      currentColumn++;
      if (currentColumn >= 20)
         currentColumn = 0;
   }
   case 2 -> {  // move down
      currentRow++;
      if (currentRow >= 16)
         currentRow = 0;
   }
   case 3 -> {  // move left
      currentColumn--;
      if (currentColumn < 0)
         currentColumn = 19;
   }
}
```

4.7.3 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. You can find a copy of the source code in RandomMosaicWalk.java. Remember that this program actually depends on two other files, Mosaic.java and MosaicCanvas.java.

```
/**
 * 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(16,20,25,25);
        fillWithRandomColors();
        currentRow = 8;   // start at center of window
        currentColumn = 10;
        while ( true ) {
            changeToRandomColor(currentRow,
currentColumn);
            randomMove();
            Mosaic.delay(10);  // Remove this line to
speed things up!
        }
    }  // 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() {
        int row, column;
        for ( row=0; row < 16; row++ ) {
            for ( 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 = (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

    /**
     * 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 is
```

```
outside the mosaic;
                  currentRow = 15;   // move it to the
opposite edge.
            }
            case 1 -> {  // move right
               currentColumn++;
               if (currentColumn >= 20)
                  currentColumn = 0;
            }
            case 2 -> {  // move down
               currentRow++;
               if (currentRow >= 16)
                  currentRow = 0;
            }
            case 3 -> {  // move left
               currentColumn--;
               if (currentColumn < 0)
                  currentColumn = 19;
            }
         }
    }  // end randomMove

} // end class RandomMosaicWalk
```

Section 4.8

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 from now on.

4.8.1 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);
}
```

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);
   }
}
```

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 {
   private static double interestRate = 0.05;
   private 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 {
   private 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;`".

Even array variables can be initialized. An array contains several elements, not just a single value. To initialize an array variable, you can provide a list of values, separated by commas, and enclosed between a pair of braces. For example:

```
int[] smallPrimes = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29
};
```

In this statement, an array of int of length 10 is created and filled with the values in the list. The length of the array is determined by the number of items in the list.

Note that this syntax for initializing arrays **cannot** be used in assignment statements. It can only be used in a declaration statement at the time when the array variable is declared.

It is also possible to initialize an array variable with an array created using the `new` operator (which **can** also be used in assignment statements). For example:

```
String[] nameList = new String[100];
```

but in that case, of course, all the array elements will have their default value.

4.8.2 Declaring Variables with var

In Java 11 and later, there is a new way of declaring variables, using the word "var" instead of specifying an explicit type for the variable. The new syntax for declarations can only be used for local variables, that is for variables that are declared inside subroutines (see Subsection 4.2.4). Furthermore a variable that is declared using `var` must be given an initial value. A variable that is declared with `var` has a defined type, just like any other variable. The Java compiler uses the type of the initial value to define the type for the variable. For example, the declaration statement

```
var interestRate = 0.05;
```

can be used to define a local variable named `interestRate` with initial value 0.05. The variable is of type double, since 0.05 is a value of type double. And a local variable named `nameList` of type `String[]` can be declared as

```
var nameList = new String[100];
```

In particular, `var` can be used to declare the loop control variable in a `for` loop. For example,

```
for ( var i = 0;  i < 10;  i++ ) {
   System.out.println(i);
}
```

All this might not seem particularly useful, but it becomes more useful for the more complicated "parameterized types" that will be covered in Section 7.3 and Chapter 10.

4.8.3 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 that 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

```
public 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. (A "final" modifier on a public interest rate makes a lot of sense—a bank might want to publish its interest rate, but it certainly wouldn't want to let random people make changes to it!)

It is legal to apply the `final` modifier to local variables and to formal parameters (and even to classes and subroutines), but it is probably 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

```
public 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`.

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 ALIGNMENT_RIGHT = 1;
public static final int ALIGNMENT_CENTER = 2;
```

In fact, this is how things had to be done before the introduction of enumerated types, and it is what is still done in many cases. 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 main 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 = 20;        // 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 red italic. Note in particular how the constants `ROWS` and `COLUMNS` are used in `randomMove()` when moving the disturbance from one edge of the mosaic to the opposite edge. I've left out most of the comments to save space.

```
public class RandomMosaicWalk2 {

    final static int ROWS = 20;        // 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 ( true ) {
            changeToRandomColor(currentRow,
currentColumn);
            randomMove();
            Mosaic.delay(5);
        }
    }  // 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;
            }
            case 1 -> {  // move right
               currentColumn++;
               if (currentColumn >= COLUMNS)
                  currentColumn = 0;
            }
            case 2 -> {  // move down
               currentRow++;
               if (currentRow >= ROWS)
                  currentRow = 0;
            }
            case 3 -> {  // move left
               currentColumn--;
               if (currentColumn < 0)
                  currentColumn = COLUMNS - 1;
            }
         }
     }  // end randomMove

} // end class RandomMosaicWalk2
```

4.8.4 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 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 Section 9.1. If the subroutine is not `private`, it can also be accessed from outside the class where it is defined, using its full name.

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, 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!

Programming Exercises for Chapter 4

Exercise 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 from Exercise 3.4 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.

See the Solution

Exercise 4.2:

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.

See the Solution

Exercise 4.3:

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.)

See the Solution

Exercise 4.4:

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
       .                  .
       .                  .
```

See the Solution

Exercise 4.5:

This exercise asks you to write a few lambda expressions and a function that returns a lambda expression as its value. Suppose that a function interface *ArrayProcessor* is defined as

```
public interface ArrayProcessor {
    double apply( double[] array );
}
```

Write a class that defines four `public static final` variables of type *ArrayProcessor* that process an array in the following ways: find the maximum value in the array, find the minimum value in an array, find the sum of the values in the array, and find the average of the values in the array. In each case, the value of the variable should be given by a lambda expression. The class should also define a function

```
public static ArrayProcessor counter( double value ) { ...
```

This function should return an *ArrayProcessor* that counts the number of times that *value* occurs in an array. The return value should be given as a lambda expression.

The class should have a *main*() routine that tests your work. The program that you write for this exercise will need access to the file ArrayProcessor.java, which defines the functional interface.

See the Solution

Exercise 4.6:

The sample program RandomMosaicWalk.java from Section 4.7 shows a "disturbance" that wanders around a grid of colored squares. When the disturbance visits a square, the color of that square is changed. Here's an idea for a variation on that program. In the new version, 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. The result will be a visually interesting effect, as the path followed by the disturbance gradually turns a brighter and brighter green.

Write a subroutine that will add 25 to the green component of one of the squares in the mosaic. (But don't let the green component go over 255, since that's the largest legal value for a color component.) 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 4.8 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.

By default, the rectangles in the mosaic have a "3D" appearance and a gray border that makes them look nicer in the random walk program. But for this program, you want to turn off that effect. To do so, call `Mosaic.setUse3DEffect(false)` in the main program.

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.

See the Solution

Exercise 4.7:

For this exercise, you will do something even more interesting with the *Mosaic* class that was discussed in Section 4.7. (Again, don't forget that you will need Mosaic.java and MosaicCanvas.java to compile and run your program.)

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.

See the Solution

Exercise 4.8:

Write a program that administers a basic addition quiz to the user. There should be ten questions. Each question is a simple addition problem such as `17 + 42`, where the numbers in the problem are chosen at random (and are not too big). The program should ask the user all ten questions and get the user's answers. After asking all the questions, the user should print each question again, with the user's answer. If the user got the answer right, the

program should say so; if not, the program should give the correct answer. At the end, tell the user their score on the quiz, where each correct answer counts for ten points.

The program should use three subroutines, one to create the quiz, one to administer the quiz, and one to grade the quiz. It can use three arrays, with three global variables of type `int[]`, to refer to the arrays. The first array holds the first number from every question, the second holds the second number from every questions, and the third holds the user's answers.

See the Solution

Quiz on Chapter 4

Question 1:

A "black box" has an interface and an implementation. Explain what is meant by the terms *interface* and *implementation*.

Question 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?

Question 3:

Briefly explain how subroutines can be useful in the top-down design of programs.

Question 4:

Discuss the concept of *parameters*. What are parameters for? What is the difference between *formal parameters* and *actual parameters*?

Question 5:

Give two different reasons for using named constants (declared with the `final` modifier).

Question 6:

What is an API? Give an example.

Question 7:

What might the following expression mean in a program?

```
(a,b) -> a*a + b*b + 1
```

Question 8:

Suppose that *SupplyInt* is a functional interface that defines the method `public int get()`. Write a lambda expression of type *SupplyInt* that gets a random integer in the range 1 to 6 inclusive. Write another lambda expression of type *SupplyInt* that gets an int by asking the user to enter an integer and then returning the user's response.

Question 9:

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

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

Question 10:

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.

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

Question 11:

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.

Question 12:

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.

Question 13:

Write a function that finds the average of the first N elements of an array of type double. The array and N are parameters to the subroutine.

Question 14:

Explain the purpose of the following function, and explain how it works:

```
static int[] stripZeros( int[] list ) {
    int count = 0;
    for (int i = 0; i < list.length; i++) {
        if ( list[i] != 0 )
            count++;
    }
    int[] newList;
    newList = new int[count];
    int j = 0;
    for (int i = 0; i < list.length; i++) {
        if ( list[i] != 0 ) {
            newList[j] = list[i];
            j++;
        }
    }
    return newList;
}
```

See the Answers

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.

The first four sections of this chapter introduce the basic things you need to know to work with objects and to define simple classes. The remaining sections cover more advanced topics; you might not understand them fully the first time through. In particular, Section 5.5 covers the most 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.

Contents of Chapter 5:

- Section 1: Objects, Instance Methods, and Instance Variables
- Section 2: Constructors and Object Initialization
- Section 3: Programming with Objects
- Section 4: Programming Example: Card, Hand, Deck
- Section 5: Inheritance, Polymorphism, and Abstract Classes
- Section 6: this and super
- Section 7: Interfaces
- Section 8: Nested Classes
- Programming Exercises
- Quiz on This Chapter

Section 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.

As I have mentioned before, in the context of object-oriented programming, subroutines are often referred to as **methods**. Now that we are starting to use objects, I will be using the term "method" more often than "subroutine."

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 methods (that is, subroutines). If an object is also a collection of variables and methods, 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 3.9, 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 methods 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`. When the class is loaded into the computer, there is a section of memory devoted to the class, and that section of memory includes space for the values of the variables `name` and `age`. We can picture the class in memory as looking like this:

An important point is that the static member variables are part of the representation of the class in memory. Their full names, `UserData.name` and `UserData.age`, use the name of the class, since they are part of the class. When we use class *UserData* to represent the user of the program, there can only be **one** user, since we only have memory space to store data about one user. Note that 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 some non-static variables:

```
class PlayerData {
   static int playerCount;
   String name;
   int age;
}
```

I've also included a static variable in the *PlayerData* class. Here, the static variable `playerCount` is stored as part of the representation of the class in memory. Its full name is `PlayerData.playerCount`, and there is only one of it, which exists as long as the program runs. However, the other two variables in the class definition are non-static. There is no such variable as `PlayerData.name` or `PlayerData.age`, since non-static variables do not become part of the class itself. But the *PlayerData* class can be used to create objects. There can be many objects created using the class, and each one will have its **own** variables called `name` and `age`. This is what it means for the non-static parts of the class to be a template for objects: Every object gets its own copy of the non-static part of the class. We can visualize the situation in the computer's memory after several objects have been created like this:

Note that the static variable `playerCount` is part of the class, and there is only one copy. On the other hand, every object contains a `name` and an `age`. An object that is created from a class is called an **instance** of that class, and as the picture shows, every object "knows" which class was used to create it. I've shown class *PlayerData* as containing something called a "constructor;" the constructor is a subroutine that creates objects.

Now there can be many "players," because we can make new objects to represent new players on demand. A program might use the *PlayerData* 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! "Dynamic" is the opposite of "static."

An object that is created using a class is said to be an **instance** of that class. We will sometimes say that the object **belongs** to the class. The variables that the object contains are called **instance variables**. The methods (that is,

subroutines) that the object contains are called **instance methods**. 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.

My examples here don't include any methods, but methods work similarly to variables. Static methods are part of the class; non-static, or instance, methods become part of objects created from the class. It's not literally true that each object contains its own copy of the actual compiled code for an instance method. But logically an instance method is part of the object, and I will continue to say that the object "contains" the instance method.

Note that you should distinguish between the **source code** for the class, and the **class itself** (in memory). 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 (in the computer's memory), 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.

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, and we will see only a few examples of classes that contain a mixture of the two.

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 test grades 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, or 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. (In fact, the `new` operator calls a special subroutine called a "constructor" in the class.) 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. The object itself is somewhere in the heap. 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. If I ever say something like "std **is** an object," you should read it as meaning "std is a variable that refers to an object."

So, suppose that the variable `std` refers to an object that is an instance of class `Student`. That object contains 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. It is **not** correct to say that the variable "points to null"; in fact, the variable **is** null. For example, you can 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:

In this picture, when a variable contains a reference to an object, the value of that variable is shown as an arrow pointing to the object. Note, by the way, that the *Strings* are objects! 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 that you are comparing are references to objects; they are not objects. So, you are testing whether `std1` and `std2` refer to the same object. This is fine, if it's 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. (Strings are special objects, treated by Java in a special way, and I haven't attempted to show the actual internal structure of the *String* objects.) Since strings are objects, 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; 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, the 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;                                  s.name =
"Fred";
}                                       }

The lines:                              The lines:

   x = 17;                                 stu.name =
```

```
"Jane";
   dontChange(x);                                change(stu);
   System.out.println(x);
System.out.println(stu.name);

output the value 17.                         output the value
"Fred".

The value of x is not                        The value of stu
is not
changed by the subroutine,                   changed, but
stu.name is changed.
which is equivalent to                       This is equivalent
to

   z = x;                                    s = stu;
   z = 42;                                   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. (Sometimes for boolean variables, "is" is used in place of "get". For example, a getter for a boolean member variable named `done` might be called `isDone()`.)

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 people, 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`.

5.1.4 Arrays and Objects

As I noted in Subsection 3.8.1, arrays are objects. Like *Strings* they are special objects, with their own unique syntax. An array type such as `int[]` or `String[]` is actually a class, and arrays are created using a special version of the `new` operator. As in the case for other object variables, an array variable can never hold an actual array—only a reference to an array object. The array object itself exists in the heap. It is possible for an array variable to hold the value `null`, which means there is no actual array.

For example, suppose that `list` is a variable of type `int[]`. If the value of `list` is `null`, then any attempt to access `list.length` or an array element `list[i]` would be an error and would cause an exception of type *NullPointerException*. If `newlist` is another variable of type `int[]`, then the assignment statement

```
newlist = list;
```

only copies the reference value in `list` into `newlist`. If `list` is `null`, the result is that `newlist` will also be `null`. If `list` points to an array, the assignment statement does **not** make a copy of the array. It just sets `newlist` to refer to the same array as `list`. For example, the output of the following code segment

```
list = new int[3];
list[1] = 17;
newlist = list; // newlist points to the same array as
list!
newlist[1] = 42;
System.out.println( list[1] );
```

would be 42, not 17, since `list[1]` and `newlist[1]` are just different names for the same element in the array. All this is very natural, once you understand that arrays are objects and array variables hold pointers to arrays.

This fact also comes into play when an array is passed as a parameter to a subroutine. The value that is copied into the subroutine is a pointer to the array. The array is not copied. Since the subroutine has a reference to the original array, any changes that it makes to elements of the array are being made to the original and will persist after the subroutine returns.

Arrays are objects. They can also hold objects. The base type of an array can be a class. We have already seen this when we used arrays of type `String[]`, but any class can be used as the base type. For example, suppose *Student* is the class defined earlier in this section. Then we can have arrays of type `Student[]`. For an array of type `Student[]`, each element of the array is a variable of type *Student*. To store information about 30 students, we could use an array:

```
Student[] classlist;  // Declare a variable of type
Student[].
classlist = new Student[30];  // The variable now points
to an array.
```

The array has 30 elements, `classlist[0]`, `classlist[1]`, ... `classlist[29]`. When the array is created, it is filled with the default initial value, which for an object type is `null`. So, although we have 30 array elements of type *Student*, we don't yet have any actual *Student* objects! All we have is 30 nulls. If we want student objects, we have to create them:

```
Student[] classlist;
classlist = new Student[30];
for ( int i = 0; i < 30; i++ ) {
    classlist[i] = new Student();
}
```

Once we have done this, each `classlist[i]` points to an object of type *Student*. If we want to talk about the name of student number 3, we can use `classlist[3].name`. The average for student number `i` can be computed by calling `classlist[i].getAverage()`. You can do anything with `classlist[i]` that you could do with any other variable of type *Student*.

Section 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, every time a new *PairOfDice* is created, 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, 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 a constructor 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. In this version of *PairOfDice*, 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.

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.
```

```
   public Student(String theName) {
        // Constructor for Student objects;
        // provides a name for the Student.
        // The name can't be null.
      if ( theName == null )
          throw new IllegalArgumentException("name can't
be null");
      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`. Furthermore, the constructor makes it impossible to have a student object whose name is `null`. 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.

Note that it would be legal, and good style, 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.

Let's take this example a little farther to illustrate one more aspect of classes: What happens when you mix static and non-static in the same class? In that case, it's legal for an instance method in the class to use static member variables or call static member subroutines. An object knows what class it belongs to, and it can refer to static members of that class. But there is only one copy of the static member, in the class itself. Effectively, all the objects share one copy of the static member.

As an example, consider a version of the *Student* class to which 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.
   public double test1, test2, test3;    // Grades on three
tests.

   private int ID;  // Unique ID number for this student.

   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() {
        // Getter method for reading the value of the
private
        // instance variable, name.
      return name;
   }

   public int getID() {
        // Getter 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.)

5.2.3 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 some 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.

Section 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. 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 *StringBuilder* 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 *StringBuilder* class. For example:

```
StringBuilder builder = new StringBuilder();
```

(This statement both declares the variable `builder` and initializes it to refer to a newly created *StringBuilder* object. Combining declaration with initialization was covered in Subsection 4.8.1 and works for objects just as it does for primitive types.)

Like a *String*, a *StringBuilder* contains a sequence of characters. However, it is possible to add new characters onto the end of a `StringBuilder` without continually making copies of the data that it already contains. If `x` is a value of any type and `builder` is the variable defined above, then the command `builder.append(x)` will add `x`, converted into a string representation, onto the end of the data that was already in the builder. This can be done more efficiently than copying the data every time something is appended. A long string can be built up in a *StringBuilder* using a sequence of `append()` commands. When the string is complete, the function `builder.toString()` will return a copy of the string in the builder as an ordinary value of type *String*. The *StringBuilder* 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 study such collection classes extensively in Chapter 10. And we have already encountered `java.util.Scanner` in Subsection 2.4.6. 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 4.6.3 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.

Some of Java's standard classes are used in GUI programming. You will encounter many of them in Chapter 6. Here, I will mention only the class *Color*, from the package `javafx.scene.paint`, so that I can use it in the next example. A *Color* object represents a color that can be used for drawing. In Section 3.9, you encountered color constants such as `Color.RED`. These constants are final static member variables in the *Color* class, and their values are objects of type *Color*. It is also possible to create new color objects. Class *Color* has a constructor `new Color(r,g,b,a)`, which takes four double parameters to specify the red, green, and blue components of the color, plus an "alpha" component that says how transparent the color is. The parameters must be in the range 0.0 to 1.0. For example, a value of 0.0 for *r* means that the color contains no red, while a value of 1.0 means that the color contains the maximum possible amount of red. When you draw with a partially transparent color, the background shows through the color to some

extent. A larger value of the fourth parameter gives a color that is less transparent and more opaque.

A *Color* object has only a few instance methods that you are likely to use. Mainly, there are functions like `getRed()` to get the individual color components of the color. There are no "setter" methods to change the color components. In fact, a *Color* is an **immutable** object, meaning that all of its instance variables are `final` and cannot be changed after the object is created. *Strings* are another example of immutable objects, and we will make some of our own later in this chapter.

The main point of all this, 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 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*, in package `java.lang`. (*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 two of them here. You will encounter more of them later in the book.

The method `equals(obj)` is defined in class *Object*. It takes one parameter, which can be any object. It is meant for testing whether two objects are "equal" but its definition gives `obj1.equals(obj2)` the same meaning as

`obj1 == obj2`. That is, it tests whether `obj1` and `obj2` refer to the same object. The *String* class overrides this method to say that two *Strings* are equal if they contain the same sequence of characters, and it is common to similarly override `equals()` in a class to say that two objects belonging to that class are equal if they have the same contents. We see again that what it means for objects to be "equal" is not always clear. We will have more use for this method later.

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.

5.3.3 Writing and Using a Class

As an example of designing and using a new class, we will write an animation program, based on the same animation framework that was used in Subsection 3.9.3. The animation shows a number of semi-transparent disks that grow in size as the animation plays. The disks have random colors and locations. When a disk gets too big, or sometimes just at random, the disk disappears and is replaced with a new disk at a random location. Here is a reduced-size screenshot from the program:

A disk in this program can be represented as an object. A disk has properties —color, location, and size—that can be instance variables in the object. As for instance methods, we need to think about what we might want to do with a disk. An obvious candidate is that we need to be able to draw it, so we can include an instance method `draw(g)`, where `g` is a graphics context that will be used to do the drawing. The class can also include one or more constructors. A constructor initializes the object. It's not always clear what data should be provided as parameters to the constructor. In this case, as an example, the constructor's parameters specify the location and size for the

circle, but the constructor makes up a color using random values for the red, green, and blue components. Here's the complete class:

```
import javafx.scene.paint.Color;
import javafx.scene.canvas.GraphicsContext;

/**
 * A simple class that holds the size, color, and location
of a colored disk,
 * with a method for drawing the circle in a graphics
context.  The circle
 * is drawn as a filled oval, with a black outline.
 */
public class CircleInfo {

    public int radius;    // The radius of the circle.
    public int x,y;       // The location of the center of
the circle.
    public Color color;   // The color of the circle.

    /**
     * Create a CircleInfo with a given location and
radius and with a
     * randomly selected, semi-transparent color.
     * @param centerX   The x coordinate of the center.
     * @param centerY   The y coordinate of the center.
     * @param rad       The radius of the circle.
     */
    public CircleInfo( int centerX, int centerY, int rad )
{
        x = centerX;
        y = centerY;
        radius = rad;
        double red = Math.random();
        double green = Math.random();
        double blue = Math.random();
        color = new Color( red,green,blue, 0.4 );
    }

    /**
     * Draw the disk in graphics context g, with a black
outline.
     */
    public void draw( GraphicsContext g ) {
        g.setFill( color );
        g.fillOval( x - radius, y - radius, 2*radius,
2*radius );
```

```
        g.setStroke( Color.BLACK );
        g.strokeOval( x - radius, y - radius, 2*radius,
2*radius );
    }
}
```

It would probably be better style to write getters and setters for the instance variables, but to keep things simple, I made the variables public.

The main program for my animation is a class *GrowingCircleAnimation*. The program uses 100 disks, each one represented by an object of type *CircleInfo*. To make that manageable, the program uses an array of objects. The array variable is an instance variable in the class:

```
private CircleInfo[] circleData; // holds the data for all
100 circles
```

Note that it is not `static`. GUI programming generally uses objects rather than static variables and methods. Basically, this is because we can imagine having several *GrowingCircleAnimations* going on at the same time, each with its own array of disks. Each animation would be represented by an object, and each object will need to have its own `circleData` instance variable. If `circleData` were static, there would only be one array and all the animations would show exactly the same thing.

The array must be created and filled with data. The array is created using `new CircleInfo[100]`, and then 100 objects of type *CircleInfo* are created to fill the array. The new objects are created with random locations and sizes. In the program, this is done before drawing the first frame of the animation. Here is the code, where `width` and `height` are the size of the drawing area:

```
circleData = new CircleInfo[100];  // create the array

for (int i = 0; i < circleData.length; i++) { // create
the objects
    circleData[i] = new CircleInfo(
                            (int)(width*Math.random()),
                            (int)(height*Math.random()),
                            (int)(100*Math.random()) );
}
```

In each frame, the radius of the disk is increased and the disk is drawn using the code

```
circleData[i].radius++;
circleData[i].draw(g);
```

These statements look complicated, so let's unpack them. Now, `circleData[i]` is an element of the array `circleData`. That means that it is a variable of type *CircleInfo*. This variable refers to an object of type *CircleInfo*, which contains a public instance variable named `radius`. This means that `circleData[i].radius` is the full name for that variable. Since it is a variable of type int, we can use the `++` operator to increment its value. So the effect of `circleData[i].radius++` is to increase the radius of the circle by one. The second line of code is similar, but in that statement, `circleData[i].draw` is an instance method in the *CircleInfo* object. The statement `circleData[i].draw(g)` calls that instance method with a parameter `g` that represents the graphics context that is being used for drawing.

The source code example GrowingCircleAnimation.java contains the full source code for the program, if you are interested. Since the program uses class *CircleInfo*, you will also need a copy of CircleInfo.java in order to compile and run the program.

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 only 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. (This is not a software engineering textbook.) 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.

Section 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, 3, ..., 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 at least 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. You can look at the source code, Deck.java, if you want. It should not be a surprise that the class includes an array of *Cards* as an instance variable, but there are a few things you might not understand at this point. Of course, you can use the class in your programs as a black box, 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, there are a few things in the implementation of the class that you won't understand at this point, but that doesn't stop you from using the class in your projects. The source code can be found in the file Hand.java

---

5.4.2 The Card Class

We will look at the design and implementation of a *Card* class in full detail. 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 ints`. It might be better to use an enumerated type, but I will stick here to integer-valued constants.) 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. Since all its instance variables are `final`, a *Card* is an immutable object.

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, which can also be found in Card.java. This class 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 { // (Note use of traditional syntax for the
switch statement.)
         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. 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:

```
import textio.TextIO;

/**
 * This program lets the user play HighLow, a simple card
game
 * that is described in the output statement 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,
          HighLow.  A card is dealt from a deck of cards.
          You have to predict whether the next card will
be
          higher or lower.  Your score in the game is the
          number of correct predictions you make before
          you guess wrong.
          """);

      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++;
         System.out.print("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 in 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();
      System.out.println("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'). */

         System.out.print("Will the next card be higher
(H) or lower (L)?  ");
         do {
             guess = TextIO.getlnChar();
             guess = Character.toUpperCase(guess);
             if (guess != 'H' && guess != 'L')
                System.out.print("Please respond with H or
L:  ");
         } while (guess != 'H' && guess != 'L');

         /* Get the next card and show it to the user. */

         nextCard = deck.dealCard();
         System.out.println("The next card is " +
nextCard);

         /* Check the user's prediction. */

         if (nextCard.getValue() ==
currentCard.getValue()) {
            System.out.println("The value is the same as
the previous card.");
            System.out.println("You lose on ties.
Sorry!");
            break;  // End the game.
         }
         else if (nextCard.getValue() >
currentCard.getValue()) {
            if (guess == 'H') {
                System.out.println("Your prediction was
correct.");
                correctGuesses++;
            }
            else {
                System.out.println("Your prediction was
incorrect.");
                break;  // End the game.
            }
         }
         else {  // nextCard is lower
            if (guess == 'L') {
                System.out.println("Your prediction was
correct.");
                correctGuesses++;
            }
            else {
```

```
                System.out.println("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;
        System.out.println();
        System.out.println("The card is " + currentCard);

      } // end of while loop

      System.out.println();
      System.out.println("The game is over.");
      System.out.println("You made " + correctGuesses
                                          + " correct
predictions.");
      System.out.println();

      return correctGuesses;

   }  // end play()


} // end class HighLow
```

Section 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

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 they are likely to be defined in separate files and declared as `public` classes):

```
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;  // ERROR!
```

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 ints and shorts, 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 be read as `(Car)(myVehicle.numberOfDoors)`, an attempt 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 myCar;
   myCar = (Car)myVehicle;  // Type-cast to get access to
numberOfDoors!
   System.out.println("Number of doors:  " +
myCar.numberOfDoors);
}
else if (myVehicle instanceof Truck) {
   System.out.println("Type of vehicle:  Truck");
   Truck myTruck;
   myTruck = (Truck)myVehicle;  // Type-cast to get access
to numberOfAxles!
   System.out.println("Number of axles:  " +
myTruck.numberOfAxles);
}
else if (myVehicle instanceof Motorcycle) {
   System.out.println("Type of vehicle:  Motorcycle");
   Motorcycle myCycle;
   myCycle = (Motorcycle)myVehicle;  // Type-cast to get
access to hasSidecar!
   System.out.println("Has a sidecar:    " +
```

```
myCycle.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. The code above avoids *ClassCastExceptions* by using `instanceof` to test the type of the variable before doing a type cast.

In Java 17, the previous example can also be written using one of the more obscure new language features, known as **pattern matching** for `instanceof`. Pattern matching makes it possible to include declaration and initialization of a variable in an `instanceof` test. For example,

```
if (myVehicle instanceof Car myCar) {
    System.out.println("Type of vehicle: Car");
    System.out.println(Number of doors: " +
myCar.numberOfDoors);
}
```

If the test succeeds, then the variable `myCar` is created and automatically assigned the value `(Car)myVehicle`. The scope of the variable is limited to the body of the `if` statement.

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.)

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 values of those properties. 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; // (must be imported from package
javafx.scene.paint)

    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()` command in the `setColor()` method? 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
}
```

Suppose that `someShape` is a variable of type *Shape*. Then it could refer to an object of any of the types *Rectangle*, *Oval*, or *RoundRect*. As a program executes, and the value of `someShape` changes, it could even refer to objects of different types at different times! Whenever the statement

```
someShape.redraw();
```

is executed, the redraw method that is actually called is the one appropriate for the type of object to which `someShape` 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 `someShape` 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 `someShape` changes

as the loop is executed, it is possible that the very same statement "`someShape.redraw();`" will call different methods and draw different kinds of shapes as it is executed several times. 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 at run time. Polymorphism is one of the major distinguishing features of object-oriented programming. This can be seen most vividly, perhaps, if we have an array of shapes. Suppose that `shapelist` is a variable of type `Shape[]`, and that the array has already been created and filled with data. Some of the elements in the array might be *Rectangles*, some might be *Ovals*, and some might be *RoundRects*. We can draw all the shapes in the array by saying

```
for (int i = 0; i < shapelist.length; i++ ) {
    Shape shape = shapelist[i];
    shape.redraw();
}
```

As the computer goes through this loop, the statement `shape.redraw()` will sometimes draw a rectangle, sometimes an oval, and sometimes a roundrect, depending on the type of object to which array element number `i` refers.

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 "`someShape.redraw();`" is a message to the object referred to by `someShape`. 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 "`someShape.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:

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 `someShape.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 "`someShape.redraw();`", the `redraw` message is sent to the object `someShape`. 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 the `redraw()` method from the *Oval* class is executed. 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.

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 "`someShape.redraw();`". The compiler would complain that `someShape` 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 5.3.2 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*. However, there is not much that you can do with them.)

Since every class is a subclass of *Object*, a variable of type *Object* can refer to any object whatsoever, of any type. Similarly, an array of type `Object[]` can hold objects of any type.

The sample source code file ShapeDraw.java uses an abstract *Shape* class and an array of type `Shape[]` to hold a list of shapes. 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 *GraphicsContext>*. This parameter is required because all drawing in Java requires a graphics context.) 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 array is used to hold the list of shapes. Here is a scaled-down screenshot from the program:

If you run the `ShapeDraw` program, you can click one of the buttons along the bottom to add a shape to the picture. The new shape will appear in the upper left corner of the drawing area. The color of the shape is given by the "pop-up menu" of colors below the drawing area. Once a shape is on the screen, you can drag it around with the mouse. A shape will maintain the same front-to-back order with respect to other shapes on the screen, even while you are dragging it. However, you can move a shape out in front of all the other shapes if you hold down the shift key as you click on it.

In the program, 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, and program the button to add the correct type of

shape to the screen. No other changes in the programming would be necessary.

5.5.6 Final Methods and Classes

We have seen how the `final` modifier applies to variables: It makes it impossible to assign a new value to the variable after it has been initialized. (For a `final` instance variable, the initialization can take place in the constructors for the class that contains the variable.) However, the `final` modifier can also be applied to classes and to method definitions.

A `final` class cannot be used be used as a basis for making subclasses. That is, if a class is defined as "`final class A...`", then it is a syntax error to try to extend *A* with "class B extends A...".

If a method is declared `final` in a class, then it is not possible for a subclass to contain a method with the same signature. That is, the method cannot be "overridden". This is true both for `static` methods and for instance methods. Note that in a `final` class, all methods are implicitly final since it is not even possible to make a subclass of that class.

The idea is that a `final` class or method is in its final form—it cannot be modified or replaced by a subclass. With `final`, it is possible to make a guarantee about the behavior of a method. For example, suppose that a *Rectangle* class has an instance method `draw()` that draws a rectangle, and that `rect` is a variable of type *Rectangle*. If `draw()` is a `final` method, then we are guaranteed that a call to `rect.draw()` will draw a rectangle. But if `draw()` is **not** `final`, then it could draw anything, or nothing! The problem is that `rect` could actually refer to an object belonging to some subclass of *Rectangle*, and that subclass could define its own `draw()` method that overrides the `draw()` method from class *Rectangle*. Using `final` can make it easier to reason about the behavior of a program.

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. The rest of this chapter covers more of those annoying details. Remember that you don't need to master everything in this chapter the first time through. In this section, we'll look at two variables, `this` and `super`, that are automatically defined in any instance method or constructor.

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 an instance variable or instance method is actually contained in an object rather than in a class, and each object has its own version. A 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 a full name for an instance variable named `test1` that is contained in that object.

But when we are working inside a class and use a simple name to refer to an instance variable like `test1`, where is the object that contains the variable? The solution to this riddle is simple: Suppose that a reference to "`test1`" occurs in the definition of some instance method. The method is part of 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, it is not part of an object, and hence there are no instance members in sight!)

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. The variable `this` can be 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.

(Some object oriented languages use the name "self" instead of "this." Here, an object is seen as an entity that receives messages and responds by performing some action. From the point of view of that entity, an instance

variable such as `self.name` refers to the entity's own `name`, something that is part of the entity itself. Calling an instance method such as `self.redraw()` is like saying "message to self: redraw!")

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 that is also called "name." However, the instance variable can still be referred to by its full name, which is `this.name`. In the assignment statement "`this.name = name`", the "name" on the right is the formal parameter, and the value of that formal parameter is being 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);`". Or you could assign the value of `this` to another variable in an assignment statement. You can store it in an array. In fact, you can do anything with `this` that you could do with any other variable, except change its value. (Consider it to be a `final` variable.)

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 you use a method call `super.doSomething()` in a class that you are writing. Now, `super` doesn't know anything about any `doSomething()` method in the same class. It only knows about things in the superclass of that class, so `super.doSomething()` represents an attempt to execute a method named `doSomething()` from the superclass. If there is no such method in the superclass, you'll get a syntax error—even if there is a `doSomething()` method in the class that you are writing.

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 a subclass contains an instance method 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!

5.6.3 super and this As Constructors

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 even know how it is implemented. 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.)

You can use the special variable `this` in exactly the same way to call another constructor in the same class. That is, the very first line of a constructor can look like a subroutine call with "this" as the name of the subroutine. The result is that the body of another constructor in the same class is executed. This can be very useful since it can save you from repeating the same code in several different constructors. As an example, consider MosaicCanvas.java, which was used indirectly in Section 4.7. A *MosaicCanvas* represents a grid of colored rectangles. It has a constructor with four parameters:

```
public MosaicCanvas(int rows, int columns,
                    int preferredBlockWidth, int
preferredBlockHeight)
```

This constructor provides several options and does a lot of initialization. I wanted to provide easier-to-use constructors with fewer options, but all the initialization still has to be done. The class also contains these constructors:

```
public MosaicCanvas() {
    this(42,42);
}

public MosaicCanvas(int rows, int columns) {
    this(rows,columns,16,16);
}
```

Each of these constructors exists just to call another constructor, while providing constant values for some of the parameters. For example, `this(42,42)` calls the second constructor listed here, while that constructor in turn calls the main, four-parameter constructor. That main constructor is eventually called in all cases, so that all the essential initialization gets done in every case.

Section 5.7

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 have already encountered "functional interfaces" in Section 4.5 in connection with lambda expressions. A functional interface specifies a single method. However, interfaces can be much more complicated than that, and they have many other uses.

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, and you will need to learn how to use them.

5.7.1 Defining and Implementing Interfaces

We encountered the term "interface" in other contexts, 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, as we'll see later.) 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 Strokeable {
   public void stroke(GraphicsContext g);
}
```

This looks much like a class definition, except that the implementation of the `stroke()` method is omitted. A class that implements the `interface` *Strokeable* must provide an implementation for `stroke()`. Of course, the class can also include other methods and variables. For example,

```
public class Line implements Strokeable {
    public void stroke(GraphicsContext g) {
        . . . // do something—presumably, draw a line
    }
    . . . // other methods, variables, and constructors
}
```

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`** `Strokeable`". Any concrete class that implements the *Strokeable* interface must define a `stroke()` instance method. Any object created from such a class includes a `stroke()` 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 *Strokeable* 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 Strokeable, 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 *Strokeable* interface with the abstract class

```
public abstract class AbstractStrokeable {
   public abstract void stroke(GraphicsContext g);
}
```

The main difference is that a class that extends *AbstractStrokeable* cannot extend any other class, while a class that implements *Strokeable* can also extend some class, as well as implement other interfaces. Of course, an abstract class can contain non-abstract methods as well as abstract methods. An interface is like a "pure" abstract class, which contains only abstract methods.

Note that the methods declared in an interface must be `public` and `abstract`. In fact, since that is the only option, it is not necessary to specify either of these modifiers in the declaration.

In addition to method declarations, an interface can also include variable declarations. The variables must be `"public static final"` and effectively become public static final variables in every class that implements the interface. In fact, since the variables can only be public and static and final, specifying the modifiers is optional. For example,

```
public interface ConversionFactors {
    int INCHES_PER_FOOT = 12;
    int FEET_PER_YARD = 3;
    int YARDS_PER_MILE = 1760;
}
```

This is a convenient way to define named constants that can be used in several classes. A class that implements *ConversionFactors* can use the constants defined in the interface as if they were defined in the class.

Note in particular that any variable that is defined in an interface is a constant. It's not really a variable at all. An interface **cannot** add instance variables to the classes that implement it.

An interface can extend one or more other interfaces. For example, if *Strokeable* is the interface given above and *Fillable* is an interface that defines a method `fill(g)`, then we could define

```
public interface Drawable extends Strokeable, Fillable {
    // (more methods/constants could be defined here)
}
```

A concrete class that implements *Drawable* must then provide implementations for the `stroke()` method from *Strokeable* and the `fill()` method from *Fillable*, as well as for any abstract methods specified directly in the *Drawable* interface.

An `interface` is usually defined in its own .java file, whose name must match the name of the `interface`. For example, *Strokeable* would be

defined in a file named `Strokeable.java`. Just like a `class`, an `interface` can be in a package and can `import` things from other packages.

This discussion has been about the syntax rules for `interfaces`. But of course, an `interface` also has a semantic component. That is, the person who creates the `interface` intends for the methods that it defines to have some specific meaning. The `interface` definition should include comments to express that meaning, and classes that implement the `interface` should take that meaning into account. The Java compiler, however, can only check the syntax; it can't enforce the meaning. For example, the `stroke()` method in an object that implements *Strokeable* is presumably meant to draw a graphical representation of the object by stroking it, but the compiler can only check that the `stroke()` method exists in the object.

5.7.2 Default Methods

Starting in Java 8, interfaces can contain *default methods*. Unlike the usual abstract methods in interfaces, a default method has an implementation. When a class implements the interface, it does not have to provide an implementation for the default method—although it can do so if it wants to provide a different implementation. Essentially, default methods are inherited from interfaces in much the same way that ordinary methods are inherited from classes. This moves Java partway towards supporting multiple inheritance. It's not true multiple inheritance, however, since interfaces still cannot define instance variables. Default methods can call abstract methods that are defined in the same interface, but they cannot refer to any instance variables.

Note that a functional interface can include default methods in addition to the single abstract method that it specifies.

A default method in an interface must be marked with the modifier `default`. It can optionally be marked `public` but, as for everything else in interfaces, default methods are automatically public and the `public` modifier can be omitted. Here is an example:

```
public interface Readable { // represents a source of
input

    public char readChar();  // read the next character
from the input

    default public String readLine() { // read up to the
next line feed
        StringBuilder line = new StringBuilder();
        char ch = readChar();
        while (ch != '\n') {
            line.append(ch);
            ch = readChar();
        }
        return line.toString();
    }

}
```

A concrete class that implements this interface must provide an implementation for `readChar()`. It will inherit a definition for `readLine()` from the interface, but can provide a new definition if necessary. When a class includes an implementation for a `default` method, the implementation given in the class overrides the default method from the `interface`.

Note that the default `readLine()` calls the abstract method `readChar()`, whose definition will only be provided in an implementing class. The reference to `readChar()` in the definition is polymorphic. The default implementation of `readLine()` is one that would make sense in almost any class that implements *Readable*. Here's a rather silly example of a class that implements *Readable*, including a `main()` routine that tests the class. Can you figure out what it does?

```
public class Stars implements Readable {

    public char readChar() {
        if (Math.random() > 0.02)
           return '*';
        else
           return '\n';
    }

    public static void main(String[] args) {
        Stars stars = new Stars();
```

```
        for (int i = 0 ; i < 10; i++ ) {
            String line = stars.readLine();  // Calls the
default method!
            System.out.println( line );
        }
    }

}
```

Default methods provide Java with a capability similar to something called a "mixin" in other programming languages, namely the ability to mix functionality from another source into a class. Since a class can implement several interfaces, it is possible to mix in functionality from several different sources.

5.7.3 Interfaces as Types

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 *Strokeable* is the interface given above, and if *Line* and *Circle* are classes that implement *Strokeable*, as above, then you could say:

```
Strokeable figure;  // Declare a variable of type
Strokeable.  It
                    //    can refer to any object that
implements the
                    //    Strokeable interface.

figure = new Line();  // figure now refers to an object of
class Line
figure.stroke(g);   // calls stroke() method from class
Line

figure = new Circle();   // Now, figure refers to an
object
                                //   of class Circle.
figure.stroke(g);   // calls stroke() method from class
Circle
```

A variable of type *Strokeable* can refer to any object of any class that implements the *Strokeable* interface. A statement like `figure.stroke(g)`,

above, is legal because `figure` is of type *Strokeable*, and **any** *Strokeable* object has a `stroke()` method. So, whatever object `figure` refers to, that object must have a `stroke()` 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 (given a few special cases, such as an `enum`, which is considered to be a special kind of class). Of these, however, only classes can be used to construct new objects.

An interface can also be the base type of an array. For example, we can use an array type `Strokeable[]` to declare variables and create arrays. The elements of the array can refer to any objects that implement the *Strokeable* interface:

```
Strokeable[] listOfFigures;
listOfFigures = new Strokeable[10];
listOfFigures[0] = new Line();
listOfFigures[1] = new Circle();
listOfFigures[2] = new Line();
   .
   .
   .
```

Every element of the array will then have a `stroke()` method, so that we can say things like `listOfFigures[i].stroke(g)`.

Section 5.8

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. (In fact, a class can even be nested inside a method, which must, of course, itself be inside a 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. Interfaces, like classes, can be nested inside class definitions and can be either static or non-static. (In fact, interface definitions can contain static nested classes and interfaces, but that is not something that you will see in this textbook.)

5.8.1 Static Nested Classes

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 is used outside the containing class, its

name must indicate its membership in the containing class. That is, the full name of the static nested class consists of the name of the class in which it is nested, followed by a period, followed by the name of the nested 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. 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
```

The full name of the nested class is *WireFrameModel.Line*. That name can be used, for example, to declare variables. 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, even if they are marked `private`. 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`.

5.8.2 Inner Classes

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 with the class in which its definition 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 (although it does not literally contain a copy of the compiled code for 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.

In most cases, an inner class is used only within the class where it is defined. When that is true, using the inner class is really not much different from using any other class. You can create variables and declare objects using the simple name of the inner class in the usual way (although you can only do that in the non-static part of the class).

From outside the containing class, however, an inner 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 inner class. 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.)

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. The 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: it

associates any *Player* object with some particular *PokerGame* object and gives it access to the instance variables for that particular game. This is the most natural way for players to behave. A *Player* object represents a player of one particular poker game. If *Player* were an independent class or a **static** nested class, on the other hand, it would represent the general idea of a poker player, independent of a particular poker game.

5.8.3 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. At run time, it 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**.

For now, we will look at one not-very-plausible example. Suppose that *Drawable* is an interface defined as:

```
public interface Drawable {
    public void draw(GraphicsContext g);
}
```

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() {
       public void draw(GraphicsContext g) {
          g.setFill(Color.RED);
          g.fillRect(10,10,100,100);
       }
   };
```

Then `redSquare` refers to an object that implements *Drawable* and that draws a red square when its `draw()` method is called. By the way, the semicolon at the end of the statement is not part of the class definition; it's the semicolon that is required at the end of every declaration statement.

Anonymous classes are often used for actual parameters. For example, consider the following simple method, which draws a *Drawable* in two different graphics contexts:

```
void drawTwice( GraphicsContext g1, GraphicsContext g2,
Drawable figure ) {
    figure.draw(g1);
    figure.draw(g2);
}
```

When calling this method, the third parameter can be created using an anonymous inner class. For example:

```
drawTwice( firstG, secondG, new Drawable() {
          void draw(GraphicsContext g) {
             g.fillOval(10,10,100,100);
          }
     } );
```

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.

Of course, in this example, *Drawable* is a functional interface, and we could use lambda expressions (Section 4.5) instead of anonymous classes. The last example could then be written simply

```
drawTwice( firstG, secondG, g -> g.fillOval(10,10,100,100)
);
```

and `redSquare` could be defined as

```
Drawable redSquare = g -> {
          g.setFill(Color.RED);
          g.fillRect(10,10,100,100);
        };
```

This approach has the advantage that it does **not** create an extra .class file. However, lambda expressions can only be used with functional interfaces, while anonymous classes can be used with any interface or class.

5.8.4 Local Classes and Lambda Expressions

A class can be defined inside a subroutine definition. Such classes are called **local classes**. A local class can only be used inside the subroutine where it is defined. However, an object that is defined by a local class can be used outside that subroutine. It can be returned as the value of the subroutine, or it can be passed as a parameter to another subroutine. This is possible because an object belonging to some class *B* can be assigned to a variable of type *A*, as long as *B* is a subclass of *A* or, if *A* is an interface, as long as class *B* implements interface *A*. For example, if a subroutine takes a parameter of type *Drawable*, where *Drawable* is the interface defined above, then any object that implements *Drawable* can be passed as a parameter to that subroutine. And that object can be defined by a local class.

In an example earlier in this section, we passed a customized object of type *Drawable* to the `drawTwice()` method, which takes a parameter of type

Drawable. In that example, the class was an anonymous inner class. Local classes are often anonymous, but that is not required. It is also true that anonymous classes are often local classes, but that is also not required. For example, an anonymous class could be used to define the initial value of a global variable. In that case, the anonymous class is not enclosed in any subroutine and therefore is not local.

The definition of a local class can use local variables from the subroutine where it is defined. It can also use parameters to that subroutine. However, there is a restriction on the use of such variables and parameters in a local class: The local variable or parameter must be declared to be `final` or, if it is not explicitly declared `final`, then it must be "effectively final." A parameter is effectively final if its value is not changed inside the subroutine (including in any local class that references the parameter). A local variable is effectively final if its value is never changed after it is initialized. Note that there is no such restriction on global variables that are used in the definition of a local class.

The same restriction on the use of local variables also applies to lambda expressions, which are very similar to anonymous classes. Here is an example using the *FunctionR2R* functional interface from Subsection 4.5.2, which defines the single method "`double valueAt(double x))`". This code segment creates an array of *FunctionR2R* objects, where `multpliers[i]` is a function that multiplies its parameter by i:

```
FunctionR2R[] multipliers = new FunctionR2R[100];
for (int i = 0; i < 100; i++) {
   int n = i;
   multipliers[i] =  z -> n * z;
}
```

The local variable `n` is effectively final and therefore can be used in the lambda expression. On the other hand, it would have been illegal to use the variable `i` directly in the lambda expression, since `i` is not effectively final; its value is changed when `i++` is executed. Note also that this example could be written using an anonymous class instead of a lambda expression:

```
FunctionR2R[] multipliers = new FunctionR2R[100];
for (int i = 0; i < 100; i++) {
```

```
    int n = i;
    multipliers[i] =  new FunctionR2R() {
        public double valueAt(double x) {
           return n * x;
        }
    };
}
```

Programming Exercises for Chapter 5

Exercise 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, including at least a `toString()` method. 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.

See the Solution

Exercise 5.2:

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 instance methods are available:

- `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`.

See the Solution

Exercise 5.3:

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 required.)

See the Solution

Exercise 5.4:

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()`, which 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.

See the Solution

Exercise 5.5:

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.

See the Solution

Exercise 5.6:

Exercise 4.8 asked you to write a program that administers a 10-question addition quiz. Rewrite that program so that it uses the following class to represent addition questions:

```
public class AdditionQuestion {

    private int a, b;  // The numbers in the problem.

    public AdditionQuestion() { // constructor
        a = (int)(Math.random() * 50 + 1);
        b = (int)(Math.random() * 50);
    }

    public String getQuestion() {
        return "What is " + a + " + " + b + " ?";
    }
```

```
    public int getCorrectAnswer() {
        return a + b;
    }

}
```

See the Solution

Exercise 5.7:

Rewrite the program from the previous exercise so that it administers a quiz with several different kinds of questions. In the previous exercise, you used a class to represent addition questions. For this exercise, you will use the following `interface`, or an equivalent abstract class, to represent the more general idea of a question that has an integer as its answer:

```
public interface IntQuestion {
    public String getQuestion();
    public int getCorrectAnswer();
}
```

You can make the *AdditionQuestion* class implement the interface simply by adding "`implements IntQuestion`" to its definition. Write a similar class to represent subtraction questions. When creating a subtraction problem, you should make sure that the answer is not negative.

For the new program, use an array of type `IntQuestion[]` to hold the quiz questions. Include some addition questions and some subtraction questions in the quiz. You can also add a couple non-math questions, including this one, created as an anonymous class:

```
IntQuestion bigQuestion = new IntQuestion() {
    public String getQuestion() {
        return "What is the answer to the ultimate
question " +
                  " of life, the universe, and
everything?";
    }
    public int getCorrectAnswer() {
        return 42;
```

```
        }
};
```

See the Solution

Quiz on Chapter 5

Question 1:

Object-oriented programming uses *classes* and *objects*. What are classes and what are objects? What is the relationship between classes and objects?

Question 2:

Explain carefully what *null* means in Java, and why this special value is necessary.

Question 3:

What is a *constructor?* What is the purpose of a constructor in a class?

Question 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.)

Question 5:

What is meant by the terms *instance variable* and *instance method*?

Question 6:

Explain what is meant by the terms *subclass* and *superclass*.

Question 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;
}
```

Question 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.

Question 9:

Explain the term *polymorphism.*

Question 10:

Java uses "garbage collection" for memory management. Explain what is meant here by garbage collection. What is the alternative to garbage collection?

Question 11:

What is an *abstract class*, and how can you recognize an abstract class in Java?

Question 12:

What is `this`?

Question 13:

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`.

Question 14:

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.");
```

Question 15:

Explain why it can **never** make sense to test "`if (obj.equals(null))`".

See the Answers

Chapter 6

Introduction to GUI Programming

COMPUTER USERS TODAY EXPECT to interact with their computers using a graphical user interface (GUI), and Java can be used to write sophisticated GUI programs.

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. The purpose of this chapter is, as much as anything, to give you some experience with a large-scale object-oriented API.

This chapter is just an introduction to JavaFX, but it covers the essential features of GUI programming in enough detail to write some interesting programs. The discussion of JavaFX will continue in Chapter 13 with more details and with more advanced techniques, but complete coverage of JavaFX would require an entire book of its own.

Note that JavaFX is not distributed as part of the Java Development Kit. For information about how to obtain JavaFX and how to compile and run programs that use it, see Section 2.6.

This edition of this textbook covers GUI programming using the JavaFX GUI toolkit. An alternative edition covers the Swing GUI toolkit instead of JavaFX. Swing is included as a standard part of Java, so does not require any extra downloads or configuration. The Swing edition can be found at https://math.hws.edu/javanotes9-swing. The only really significant differences between the two editions are in this chapter and in Chapter 13.

Contents of Chapter 6:

- Section 1: A Basic JavaFX Application
- Section 2: Some Basic Classes
- Section 3: Basic Events
- Section 4: Basic Controls
- Section 5: Basic Layout
- Section 6: Complete Programs
- Programming Exercises
- Quiz on This Chapter

Section 6.1

A Basic JavaFX Application

THE COMMAND-LINE PROGRAMS that you have learned how to write would seem very alien to most computer users. The style of interaction where the user and the computer take turns typing strings of text seems like something out of the early days of computing, although it was only in the mid 1980s that home computers with graphical user interfaces started to become available. Today, most people interact with their computers exclusively through a GUI. A GUI program offers a much richer type of user interface, where the user uses a mouse and keyboard (or other input devices) to interact with GUI components such as windows, menus, buttons, check boxes, text input boxes, scroll bars, and so on.

This section introduces some of the basic ideas of programming with JavaFX by looking at a very simple GUI application. ("Application" is the preferred term for "program" in this context.) The application simply displays a window containing a message and three buttons. Here's what the window looks like when it first opens:

Clicking "Say Hello" will get the computer to tell you, "Hello World!". Clicking "Say Goodbye" will change the text of the message to "Goodbye".

Clicking the "Quit" button will end the application, which can also be ended by clicking the window's close box.

6.1.1 JavaFX Applications

A JavaFX program (or "application") is represented by an object of type *Application*, which is defined in the package `javafx.application`. *Application* is an abstract class, which defines, among other things, one abstract instance method, named `start()`. To create a JavaFX program, you need to create a class that extends *Application* and provides a definition for the `start`() method. (See Subsection 5.5.1 and Subsection 5.5.5.)

The class that you write to create a JavaFX application also typically includes a `main()` method that simply "launches" the application:

```
public static void main(String[] args) {
    launch();
}
```

(The call to `launch()` can be replace by `launch(args)` for an application that uses command-line arguments (Subsection 4.3.6), but the parameter is optional for all of the examples in this book.) When this `main()` routine is executed, the `launch()` method creates a new thread, called the **JavaFX application thread**. Recall from Section 1.2 that a thread can execute a sequence of instructions that can be run in parallel with other threads. It is important that anything that affects the GUI be done on the JavaFX application thread. That will be pretty much automatic for the things that we do in this chapter, but it's something that will become important when we cover threads in Chapter 12 and write some GUI programs that use several threads. The `launch()` method then creates the object that represents the application; that object is an instance of the class that contains the call to the `launch()` method. The `start()` method of that object is then called on the JavaFX application thread, and it is the responsibility of that `start()` method to set up the GUI and open a window on the screen.

Here, then is our first JavaFX application. We will spend the rest of this section discussing it:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.application.Platform;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;
import javafx.geometry.Pos;
import javafx.scene.control.Label;
import javafx.scene.control.Button;
import javafx.scene.text.Font;

public class HelloWorldFX extends Application {

    public void start(Stage stage) {

        Label message = new Label("First FX
Application!");
        message.setFont( new Font(40) );

        Button helloButton = new Button("Say Hello");
        helloButton.setOnAction( evt ->
message.setText("Hello World!") );
        Button goodbyeButton = new Button("Say Goodbye");
        goodbyeButton.setOnAction( evt ->
message.setText("Goodbye!!") );
        Button quitButton = new Button("Quit");
        quitButton.setOnAction( evt -> Platform.exit() );

        HBox buttonBar = new HBox( 20, helloButton,
goodbyeButton, quitButton );
        buttonBar.setAlignment(Pos.CENTER);
        BorderPane root = new BorderPane();
        root.setCenter(message);
        root.setBottom(buttonBar);

        Scene scene = new Scene(root, 450, 200);
        stage.setScene(scene);
        stage.setTitle("JavaFX Test");
        stage.show();

    } // end start();

    public static void main(String[] args) {
        launch();  // Run this Application.
    }

} // end class HelloWorldFX
```

The first thing that you will notice is the large number of `imports` at the start of the program, all from subpackages of the `javafx` package. A typical JavaFX program uses many classes from such packages. When I discuss a JavaFX class for the first time, I will usually mention the package that it comes from. But in any case, you can look up the class in the JavaFX API documentation. As I write this, the documentation for JavaFX 17 can be be found at https://openjfx.io/javadoc/17/.

The *HelloWorldFX* program contains a `main` method to launch the application, and it contains the required `start()` method. Of course, we will often add other methods to our application classes, to be called by `start()`. There are also a couple other methods in *Application* that can be overridden. In particular, there is an `init()`, that will be called by the system before `start()`, and a `stop()` method that is called by the system when the application is shutting down. These two methods are defined in class *Application* to do nothing. A programmer can redefine `init()` to do some initialization and `stop()` to do cleanup. However, we will rarely if ever need them. Any initialization that we need can be done in `start()`.

6.1.2 Stage, Scene, and SceneGraph

The `start()` method has a parameter of type *Stage*, from package `javafx.stage`. A *Stage* object represents a window on the computer's screen. The stage that is passed as a parameter to `start()` is constructed by the system. It represents the main window of a program, and is often referred to as the "primary stage." A program can create other windows by constructing new objects of type *Stage*.

A window is an area on the screen that can be filled with content. It can contain GUI components such as menus, buttons, and text input boxes, as well as drawing areas like those used in the graphical programs from Section 3.9. Although the primary stage is created before `start()` is called, the window does not have any content, and it is not yet visible on the screen. The `start()` method is responsible for adding content to the window and making it visible. The very last line of `start()` in the *HelloWorldFX* program, `stage.show()`, is what makes the window visible. The rest of the method

creates content, adds the content to the window, and sets various configuration options for the content and for the window itself. For example, the line

```
stage.setTitle("JavaFX Test");
```

sets the text that will appear in the title bar at the top of the window.

A stage shows a **scene**, which fills its content area and serves as a container for the GUI components in the window. A scene is represented by an object of type *Scene*. In the sample program, the statement

```
stage.setScene(scene);
```

sets the scene that will be displayed in the content area of the stage.

A scene can be filled with things called GUI **components**, such as buttons and menu bars. Each component is represented by an object belonging to a JavaFX class. For example, a push button such as the "Say Hello" button in the sample program, is represented by an object belonging to the class *Button*, from the package `javafx.scene.control`. Some components, such as the object `buttonBar` of type *HBox*, are **containers**. A container represents a region in the window that can contain other components, including other containers. So, a window contains GUI components, inside containers, which can be inside bigger containers, each represented by an object. All of these objects make up something called the **scene graph** for the window. The scene graph shows the containment relationships among all the components in the scene. For the sample program, the scene graph looks like this:

Note that this is not a class hierarchy. It does not show the relationships among the classes of the objects in the program; rather, it is a containment hierarchy that shows how the components are contained within other components on the screen. In this scene graph, `root` and `buttonBar` are containers while `message` and the three buttons are simple components.

A scene contains a single "root" component, which is a container that contains all of the other components in the scene. In the sample program, the root component is named `root` (although of course that is not required), and the root of the scene is set when the *Scene* object is constructed:

```
Scene scene = new Scene(root, 450, 200);
```

The numbers in this constructor specify the width and the height of the scene, in pixels. The numbers can be omitted, in which case the size will be computed based on the contents of the scene.

6.1.3 Nodes and Layout

Objects that can be part of a scene graph are referred to as `nodes`. Scene graph nodes must belong to one of the subclasses of *javafx.scene.Node*. Scene graph objects that can act as containers must belong to one of the subclasses of `javafx.scene.Parent`, which is a subclass of *Node*. The nodes that are contained in a parent are called **children** of that node. The root node in a scene graph must be a *Parent*.

The buttons in *HelloWorldFX* are represented by objects of type *Button*. The constructor that is used to create the button objects specifies the text that is displayed on the button. Similarly, `message` is a node of type *Label*, from package `javafx.scene.control`, whose purpose is simply to passively display a *String*. One of the properties of a *Label* object is a font, which specifies the size and style of the characters in the displayed string. The font for the text is set by calling the label's `setFont()` method. The *Font* constructor that is used in the sample program, `new Font(40)`, takes a parameter that specifies the size of the font.

Containers are *Nodes* which can have other nodes as children. The act of arranging a container's children on the screen is referred to as **layout**. Layout means setting the size and location of the components inside the container. While it is possible for a program to set the sizes and locations directly, it is more commonly done automatically by the container. Different containers implement different layout policies. For example, an *HBox* is a container that simply arranges the components that it contains in a horizontal row. In the constructor

```
HBox buttonBar = new HBox( 20, helloButton, goodbyeButton,
quitButton );
```

the first parameter specifies the size of a gap that the *HBox* will place between its children, and the remaining parameters are nodes to be added as children of the container.

A *BorderPane* is a container that implements a completely different layout policy. A *BorderPane* can contain up to five components, one in the center of the pane and up to four more placed at the top, at the bottom, to the left, and to the right of the center. In the sample program, the root of the scene is a *BorderPane* and components are added in the pane's center and bottom positions with the statements

```
root.setCenter(message);
root.setBottom(buttonBar);
```

Layout is configurable by a large number of options. The sample program has only one example of this,

```
buttonBar.setAlignment(Pos.CENTER);
```

This command centers the buttons within the *HBox*; without it, they would be shoved over to the left edge of the window. *Pos*, short for "position," is an enumerated type (see Subsection 2.3.5). JavaFX uses many enumerated types for specifying various options.

6.1.4 Events and Event Handlers

In addition to setting up the physical layout of the window, the `start()` method configures **event handling**. In *HelloWorldFX*, an event occurs when the user clicks one of the buttons. The application must be configured to respond to, or "handle," these events. Handling an event involves two objects. The event itself is represented by an object that holds information about the event. For a button click, the event is of type *ActionEvent*, and the information that it carries is the button that was clicked. The second object is of type *EventHandler*, a functional interface that defines a method `handle(evt)`, where the parameter, `evt`, is the event object. To program a response to an event, you can create a class that implements the *EventHandler* interface and provides a definition for the `handle()` method. However, since *EventHandler* is a functional interface, the handler can alternatively be specified as a lambda expression (see Section 4.5). Lambda expressions are very commonly used in JavaFX for writing event handlers, among other uses. For example, the lambda expression

```
evt -> message.setText("Hello World!")
```

represents an event handler that responds to an event by changing the text of the message to read "Hello World!". The parameter, `evt`, is the *ActionEvent* object that represents the event. In this case, the parameter is not used in the response in any way, but it still has to be there to satisfy the syntax of the lambda expression: Since the function in the *EventHandler* interface has a parameter, the lambda expression must have a parameter to match the interface.

In addition to writing the event handler, you also have to **register** the handler with the object that will produce the event. In this case, the object is `helloButton`, and the handler is registered by calling the button's `setOnAction()` method:

```
helloButton.setOnAction( evt -> message.setText("Hello
World!") );
```

Handlers for each of the other two buttons are set up in a similar way. Remember that in each case, there is an object that generates the event in response to a user action, an object that represents the event, and an event

handler that contains the code that is executed in response to the event. This diagram summarizes how it all works:

About all that still remains to be explained in the sample program is the response to a click on the "Quit" button: `Platform.exit()`. The static `exit()` method in the *Platform* class is the preferred way to programmatically end a JavaFX program. It is preferred to `System.exit()` because it cleanly shuts down the application thread and gives it an opportunity to clean up by calling the application's `stop()` method before terminating.

This section has been only a brief overview of JavaFX applications, but it has introduced many of the fundamental concepts. We will cover all of this in much greater detail in the following sections.

Section 6.2

Some Basic Classes

In this section, we will look at some basic classes, including classes representing colors, fonts, and images. We will see how these classes are used in the *GraphicsContext* API, which you already encountered in a preliminary way in Section 3.9, but they are also useful in other parts of JavaFX. There is also a brief introduction to **CSS style sheets**, which can be used to control many aspects of the visual appearance of GUI components.

6.2.1 Color and Paint

Computer color uses an **RGB color system**. That is, a color on a computer screen is specified by three numbers, called **color components**, giving the level of red, green, and blue in the color. A color is represented by an object of type *Color*, from package `javafx.scene.paint`. In JavaFX, each color component is a double value in the range 0.0 to 1.0. A *Color* object also has a fourth component in the range 0.0 to 1.0, referred as the **alpha color component**, which is generally used to represent the transparency or opaqueness of the color when it is used for drawing. When a fully opaque color (alpha component equal to 1.0) is used for drawing, the drawing color completely replaces the current color of the drawing surface. When a fully transparent color (alpha component equal to 0.0) is used for drawing, it has no effect at all. When the alpha component is between 0.0 and 1.0, the drawing color is combined with the current color to give the new color of the drawing surface, as if the original contents of the drawing surface were being viewed through colored, translucent glass. A *Color* object can be constructed by giving its red, green, blue, and alpha components; for example,

```
Color myColor = new Color( r, g, b, a );
```

where r, g, b, and a are in the range 0.0 to 1.0. However, the *Color* class also has a number of static methods for making color objects. Static methods whose job is to create objects are sometimes called **factory methods**. So instead of using the constructor, you could also say

```
Color myColor = Color.color( r, g, b, a );
```

and in the common case of a fully opaque color, with *a* equal to 1.0, you can use

```
Color myColor = Color.color( r, g, b );
```

These static factory methods are preferable to the constructor because they have the option of reusing color objects. For example, two calls to `Color.color(0.2,0.3,1.0)` might return the same *Color* object. This is OK because color objects are immutable; that is, there is no way to change a color after it has been constructed. So there is really no reason to use two different objects to represent the same color.

Your computer screen probably uses "32-bit color," which means that the color of each pixel is actually represented using just 8 bits for each of the four color components. Eight bits can represent the 256 integers in the range 0 to 255, so computer colors have traditionally been specified using integer color components in the range 0 to 255. The *Color* class has the following static method for making colors in this way:

```
Color.rgb( r, g, b )
```

where r, g, and b are integers in the range 0 to 255. There is also `Color.rgb(r,g,b,a)` where r, g, and b are ints in the range 0 to 255, and a is a double in the range 0.0 to 1.0.

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 JavaFX, the hue is given by a double value in the range 0.0 to 360.0, while

saturation and brightness are double values in the range 0.0 to 1.0. (The hue value is given in degrees, were the colors are seen as laid out along a circle, with both 0.0 and 360.0 representing pure red.) The *Color* class has static methods `Color.hsb(h,s,b)` and `Color.hsb(h,s,b,a)` for making HSB colors. 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.hsb( 360*Math.random(), 1.0, 1.0
);
```

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. The sample program SimpleColorChooser.java lets you do that. You won't understand the source code at this time, but you can run it to play with color selection or to find RGB or HSB values for the color that want.

The *Color* class also contains a large number of constants representing colors, such as `Color.RED`, `Color.BLACK`, `Color.LIGHTGRAY`, and `Color.GOLDENROD`. It might be worth mentioning that `Color.GREEN` is the fairly dark green color given by `Color.rgb(0,128,0)`; the constant representing `Color.rgb(0,255,0)` is `Color.LIME`. There is also `Color.TRANSPARENT`, which represents a fully transparent color, with all RGBA color components equal to zero.

Given a *Color*, `c`, you can find out the values of the various color components by calling functions such as `c.getRed()`, `c.getHue()`, and `c.getOpacity()`. These methods return double values in the range 0.0 to 1.0, except for `c.getHue()`, which returns a double in the range 0.0 to 360.0.

Color is a subclass of another class, *Paint*, which represents the more general idea of "something that can be used to fill and to stroke shapes." In addition to colors, there are image paints and gradient paints. I will not use these more general paints in this chapter, but they will be covered in Subsection 13.2.2. For now, you should just know that when a method has a parameter of type *Paint*, you can use a *Color*.

6.2.2 Fonts

A **font** represents a particular size and style of text. The same character will appear different in different fonts. In JavaFX, a font is represented by an object of type *Font*, from the package `javafx.scene.text`. Although the *Font* defines a couple of constructors, the best way to make a font object is with one of the static factory methods from that class.

A font has a name, which is a string that specifies a font family such as "Times New Roman." A given family can have variations such as a bold or an italic version of the font. And a font has a size, which is specified in "points," where a point should really be 1/72 inch but might in practice be equal to the size of a pixel. The most general function for making fonts can specify all of these options:

```
Font myFont = Font.font( family, weight, posture, size );
```

If the system can't match the requested font properties exactly, it will return the font that it thinks best matches the parameters. Here, `family` is a *String* that should specify a font family that is available to the program. Unfortunately, there is no set of fonts that is required to be available. "Times New Roman," "Arial," and "Verdana" are likely to work. (These are fonts that were created by Microsoft and released for free use; they are installed on many systems.) You can pass `null` as the `familyName` to use the default font family.

Font "weight" is given as an enumerated type value from the `enum` *FontWeight*. It will usually be either `FontWeight.BOLD` or `FontWeight.NORMAL`, although there are a few other values such as `FontWeight.EXTRA_BOLD`. Similarly, font "posture" is one of the constants `FontPosture.ITALIC` or `FontPosture.REGULAR`. Both *FontWeight* and *FontPosture* are from package `javafx.scene.text`.

The *Font* class has several other static functions for making fonts, which specify only a subset of the four properties family, weight, posture, and size. These include: `Font.font(size)`, `Font.font(family)`, `Font.font(family,weight,size)`, and a few others. The missing properties will have default values, which can depend on the computer

where the program is running. The static function `Font.getDefault()` returns a font that has default values for all the properties. You can call `Font.getDefault().getSize()` to find the default point size. (It's 13.0 on my computer, but might be different on yours.) Here are a few examples of making fonts:

```
Font font1 = Font.font(40);
Font font2 = Font.font("Times New Roman", FontWeight.BOLD,
24);
Font font3 = Font.font(null, FontWeight.BOLD,
FontPosture.ITALIC, 14);
```

6.2.3 Image

The term "image" refers to something like a photograph or drawing—anything that can be represented by a rectangular grid of colored pixels. Images are often stored in files. JavaFX makes it easy to load an image from a file so that it can be displayed by a program. An image is represented by an object of type *Image*, from package `javafx.scene.image`. The constructor

```
new Image( path )
```

is used to load an image from an image file. The `path` parameter is a string that specifies the location of the file. The location can be very general, such as an image on the Internet or on the user's computer, but for now I'm interested in images from **resource files**. A resource is something that is part of a program but is not code. Resources can include things like sounds, data files, and fonts, as well as images. The system can load resources for a program from the same places where it looks for the program's .class files. For a resource file in the program's top-level directory, the path to the file is simply the name of the file. If the file is in a subdirectory of the main directory, then the path includes the subdirectory name. For example, the path "images/cards.png" refers to a file named "cards.png" inside a subdirectory named "images," and "resources/sounds/beep.aiff" refers to a file named "beep.aiff" inside a directory named "sounds" that is in turn inside a directory named "resources."

There are many kinds of image files, which store the image data in a variety of formats. For JavaFX *Image* objects, you can use image files whose names end with .gif, .jpeg (or .jpg), .png, and .bmp. So, for example, if "cards.png" is a file in the top-level program directory, you can create the image object

```
Image cards = new Image( "cards.png" );
```

The image can then be displayed in a *GraphicsContext*, as we will soon see. There will be other uses for images later in this chapter.

6.2.4 Canvas and GraphicsContext

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. Every visible GUI component is drawn by coloring pixels, and every component has a coordinate system that maps (x,y) coordinates to points within the component. Most components draw themselves, but there is one JavaFX component on which you can draw anything you want by calling the appropriate methods. Such "drawing surface" components are of type *Canvas*, in package `javafx.scene.canvas`. A *Canvas* is a *Node* and so can be part of a scene graph. (However, it is not a *Parent*, so it cannot act as a container for other nodes and it cannot be the root of a scene graph. This means that even when a canvas is the only thing that you want to show in a window, it must still be placed into a container that will serve as the root of the scene graph.)

A *Canvas* appears on the screen as a rectangle made up of pixels. A position in the rectangle is specified by a pair of 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 20-pixel by 12-pixel canvas (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, and coordinates can, in fact, be numbers of type double and can refer to points inside a pixel. For example, the center of the top left pixel in a canvas has coordinates (0.5,0.5). In fact, all drawing is done using coordinates of type double.

The width and height of a *Canvas* can be specified in the constructor that is used to create the canvas object. For example, to create a tiny 20-by-12 canvas:

```
Canvas canvas = new Canvas(20,12)
```

You can query the size of a canvas by calling `canvas.getWidth()` and `canvas.getHeight()`, which return values of type double. Canvasses are usually meant to be non-resizable, but the size can be changed if necessary by calling `canvas.setWidth(w)` and `canvas.setHeight(h)`.

When a canvas is first created, it is filled with "transparent black," that is, with a color that has all RGBA components set to zero. This makes the canvas transparent: You will see whatever lies behind the canvas in the scene.

In order to draw on a canvas, you need an object of type *GraphicsContext*. Every *Canvas* has an associated *GraphicsContext*; different *GraphicsContexts* draw on different *Canvases*. You can get the graphics context for a *Canvas*, `canvas`, by calling `canvas.getGraphicsContext2D()`. For any given *Canvas*, this method will always return the same *GraphicsContext* object. Section 3.9 discussed

some of the things that can be done with a graphics context. In particular, you learned that a shape can be stroked and, if it has an interior, it can also be filled. Methods in *GraphicsContext*, `g`, that can be used for drawing include the following, where all numeric parameters are of type double:

- `g.strokeRect(x,y,w,h)` and `g.fillRect(x,y,w,h)` — Draw a rectangle with top left corner at `(x,y)`, with width `w` and with height `h`. If `w` or `h` is less than or equal to zero, nothing is drawn.
- `g.clearRect(x,y,w,h)` — Fill the same rectangle with a fully transparent color, so that whatever lies behind the rectangle will be visible through the canvas. Note that this is **not** the same as calling `g.fillRect(x,y,w,h)` with a transparent fill color; doing that has no effect at all on the contents of the rectangle.
- `g.strokeOval(x,y,w,h)` and `g.fillOval(x,y,w,h)` — Draw an oval that just fits inside the rectangle with top left corner at `(x,y)`, with width `w` and with height `h`.
- `g.strokeRoundRect(x,y,w,h,rh,rv)` and `g.fillRoundRect(x,y,w,h,rh,rv)` — Draw a rectangle with rounded corners. The rectangle has top left corner at `(x,y)`, with width `w` and with height `h`. A quarter oval is cut off each corner, where the horizontal radius of the oval is `rh` and its vertical radius is `rv`.
- `g.strokeText(str,x,y)` and `g.fillText(str,x,y)` — Draw the text of the *String* `str`. The point `(x,y)` is the left end of the baseline of the text. (A string is drawn on top of its baseline, with descenders such as the tail of a "y" extending below the baseline.) The string can contain multiple lines separated by newline ('\n') characters; `(x,y)` then gives the baseline of the first line of the string. Note that stroking text means drawing just the outlines of the characters.
- `g.strokePolygon(xcoords,ycoords,n)` and `g.fillPolygon(xcoords,ycoords,n)` — Draw a polygon, consisting of line segments connecting a sequence of points. The number of points is given by the third parameter, `n`. The first two parameters are arrays of type `double[]` containing the coordinates of the points. An extra line segment is automatically added to connect the last point back to the first. That is, the polygon connects the points `(xcoords[0],ycoords[0])`, `(xcoords[1],ycoords[1])`, ..., `(xcoords[n-1],ycoords[n-1])`, `(xcoords[0],ycoords[0])`.

- `g.strokeLine(x1,y1,x2,y2)` — Draws a line from `(x1,y1)` to `(x2,y2)`. (It's no use trying to fill a line, since it has no interior.)

The *GraphicsContext* object, `g` has a number of properties that affect drawing. When anything is drawn using `g`, the current values of the relevant properties are used. This means that changing the value of a property does not affect anything that has already been drawn; the change only applies to things drawn in the future. Each property has a setter method and a getter method. One of the properties is the *Paint* that is used for filling (which in this chapter will always be a *Color*); this property can be set by calling `g.setFill(paint)`, and you can get its current value by calling `g.getFill()`. Similarly, the *Paint* that is used for stroking can be set and get using `g.setStroke(paint)` and `g.getStroke()`, and the width of strokes can be set and get using `g.setLineWidth(w)` and `g.getLineWidth()`, where `w` is of type double. And you can set and get the font that will be used for drawing text with `g.setFont(font)` and `g.getFont()`.

Note that stroking a shape is like dragging the **center** of a pen along the outline of the shape. The size of the pen is given by the linewidth property. The stroke that is drawn extends on both sides of the actual path of the pen by an amount equal to half of the linewidth. For example, if you draw a horizontal line of width 1 with endpoints (100,100) and (300,100), half of the stroke lies above the geometric line and half lies below it. The computer might show this by blending the color of the stroke color with the current color. If you want the stroke to nicely cover complete pixels, you should actually use (100.5,100.5) and (300.5,100.5) as the coordinates of the endpoints of the line. (Whenever you draw something, you might find that for pixels that are only partially covered, the drawing color is blended with the current color instead of replacing it. This is done to decrease the jagged appearance of shapes that are made out of whole pixels, like the line and oval in the above illustration. This is known as **antialiasing**.)

It is also possible to draw an image onto a canvas, where the image is represented by an object of type *Image*. There are several methods for drawing images:

- `g.drawImage(image,x,y)` — Draws the `image` with its upper left corner at `(x,y)`, using the actual size of the image.

- `g.drawImage(image,x,y,w,h)` — Draws the `image` in the rectangle with upper left corner at `(x,y)`, with width `w`, and with height `h`. The image is stretched or shrunk to fit that rectangle if necessary.
- `g.drawImage(image, sx,sy,sw,sh, dx,dy,dh,dw)` — Draws the contents of a specified "source" rectangle in the image to a specified "destination" rectangle on the canvas. This method lets you draw just part of an image. The source rectangle has upper left corner at `(sx,sy)`, width `sw`, and height `sh`. The last four parameters specify the destination rectangle in a similar way.

It's time for a couple of actual examples. First, an example that draws some text using a variety of fonts. The program draws multiple copies of the string "Hello JavaFX" using random fonts and locations. The text is filled with random colors and stroked with a thin black stroke:

The program uses five fonts, which are created in the `start()` method using several different static factory methods from the *Font* class:

```
font1 = Font.font("Times New Roman", FontWeight.BOLD, 20);
font2 = Font.font("Arial", FontWeight.BOLD,
FontPosture.ITALIC, 28);
font3 = Font.font("Verdana", 32);
font4 = Font.font(40);
font5 = Font.font("Times New
Roman",FontWeight.BOLD,FontPosture.ITALIC,60);
```

The program defines a `draw()` method that completely redraws the content of a canvas. It is called when the canvas is first created, and it is also called when the user clicks the "Redraw" button. The method first fills the canvas with a white background, which erases the previous contents of the canvas. It then fills and strokes 25 copies of "Hello JavaFX", using a random fill color, a random position for the text, and a randomly selected font for each copy:

```
private void draw() {

    GraphicsContext g = canvas.getGraphicsContext2D();

    double width = canvas.getWidth();
    double height = canvas.getHeight();

    g.setFill( Color.WHITE );  // fill with white
background
    g.fillRect(0, 0, width, height);

    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);
            case 2 -> g.setFont(font2);
            case 3 -> g.setFont(font3);
            case 4 -> g.setFont(font4);
            case 5 -> g.setFont(font5);
        } // end switch

        // Set the color to a bright, saturated color,
with random hue.

        double hue = 360*Math.random();
        g.setFill( Color.hsb(hue, 1.0, 1.0) );

        // Select the position of the string, at random.

        double x,y;
        x = -50 + Math.random()*(width+40);
        y = Math.random()*(height+20);

        // Draw the message.
```

```
        g.fillText("Hello JavaFX",x,y);

        // Also stroke the outline of the strings with
black.

        g.setStroke(Color.BLACK);
        g.strokeText("Hello JavaFX",x,y);

    } // end for

} // end draw()
```

You can find the full source code for the program in RandomStrings.java.

The second sample program is similar, but instead of drawing random strings, it draws five playing cards dealt at random from a deck:

The deck and cards are represented using the *Deck* and *Card* classes from Section 5.4. The card images come from the file cards.png, which is a resource file for the program. The file contains a single image that contains images of all the cards, arranged in rows and columns. Here it is, shown at reduced size:

(This image is from the Gnome desktop project, http://www.gnome.org.) The image file is loaded into the program in the `start()` method simply by saying

```
cardImages = new Image("cards.png");
```

where `cardImages` is an instance variable of type *Image*. Suppose that we want to draw the card from row number R and column number C in a *GraphicsContext* `g` (where both rows and columns are numbered starting at zero). Each card in the image is 79 pixels by 123 pixels, so the card that we want has its top left corner at (79*C,123*R). If we want to place the card on the canvas with its top left corner at (x,y), we can use the third `drawImage()` method given above, which specifies a source rectangle in the image and a destination rectangle on the canvas:

```
g.drawImage( cardImages, 79*C,123*R,79,123, x,y,79,123 );
```

In the program, the card that we want to draw is given by a variable `card` of type *Card*. The row and column in the image are determined by the suit and value of the card, which are given by `card.getSuit()` and *card.getValue()*. The values returned by these functions have to be manipulated a little to get the correct row and column numbers, and the position of the card on the canvas is calculated to leave a 20-pixel gap between one card and the next.

Here is the *draw()* method from the program, which deals five random cards from a deck and draws them:

```
private void draw() {

    GraphicsContext g = canvas.getGraphicsContext2D();

    Deck deck = new Deck();
    deck.shuffle();

    double sx,sy;  // top left corner of source rect for
card in cardImages
    double dx,dy;  // corner of destination rect for card
in the canvas

    for (int i = 0; i < 5; i++) {
        Card card = deck.dealCard();
        System.out.println(card); // for testing
        sx = 79 * (card.getValue()-1);
        sy = 123 * (3 - card.getSuit());
        dx = 20 + (79+20) * i;
        dy = 20;
        g.drawImage( cardImages, sx,sy,79,123,
dx,dy,79,123 );
    }

} // end draw()
```

For the complete program, see RandomCards.java.

6.2.5 A Bit of CSS

JavaFX makes it possible to control the style—that is, the visual appearance—of components in a GUI using **CSS** (Cascading Style Sheets). CSS is one of several languages that are used to make web pages. It can control things like colors, fonts, and borders of elements of a web page. It has been adapted to play a similar role in other contexts, such as JavaFX. I do not intend to cover CSS in any detail, and anything that can be done with CSS can also be done with Java code. However, there are some things that are just easier to do with CSS; I will cover a few of them in this short section and will use

them in my programs. For people who already know CSS, a guide to its use in JavaFX is available as I write this at

https://openjfx.io/javadoc/17/javafx.graphics/javafx/scene/doc-files/cssref.html

A CSS **style rule** consists of a property and a value for that property. For example, CSS can be used to place a border around many kinds of GUI components. A border has properties with names such as `-fx-border-color` and `-fx-border-width`. (All JavaFX CSS properties have names that begin with "-fx-" to distinguish them from regular CSS properties.) A value for `-fx-border-color` can be a color name such as `red` or `lightblue`, among other formats. One color format that I will use takes the form `#RRGGBB`, where R, G, and B stand for hexadecimal digits. A two-digit hexadecimal number can represent numbers from 0 to 255. The RR, GG, and BB in `#RRGGBB` represent the red, green, and blue components of a color, each in the range 0 to 255. For example, `#FF0000` represents pure red, and `#004444` represents a dark blue-green.

For the border width, the value can be a single size, such as `3px` or `0.2cm`. The syntax is a number followed by a unit of measure, with no space between them. Here, "px" stands for "pixels," and `3px` means 3 pixels. When a width is given by a single size, the size applies to all four sides of the border. Alternatively, four sizes can be given, separated by spaces, specifying the border width on the top, right, bottom, and left, in that order. For example, a thick, blue border could be specified as

```
-fx-border-color: blue; -fx-border-width: 5px
```

and for a dark red border that is thicker on the top than on the other sides, you can use

```
-fx-border-color: #550000; -fx-border-width: 3px 1px 1px
1px
```

When several CSS rules are used together, they should be separated by semicolons.

The background color of a component can be set using `-fx-background-color` as the property name. The value is the same sort of color specification that would be used with `-fx-border-color`.

And the CSS property `-fx-padding` represents space that is left between the content of a component and its edge (or border if there is one). Like border width, padding can be specified as either a single size or a list of four sizes. For example: `-fx-padding: 8px`.

You can apply a style to a component using its `setStyle()` method, whose parameter is a *String* containing one or more CSS style rules. For example, suppose `message` is a *Label*. By default, labels have no padding or border. They can be added with a command such as

```
message.setStyle(
   "-fx-padding: 5px; -fx-border-color: black; -fx-border-
width: 1px" );
```

You can set the font for a component that displays text using the `-fx-font` property. The value for this property specifies the size and font family for the font, and can optionally be modified by a weight ("bold"), or by a style ("italic"), or both. Some examples:

```
-fx-font: 30pt "Times New Roman";
-fx-font: bold italic 18pt serif;
-fx-font: bold 42pt monospace;
```

Note that if the font family name contains spaces, it must be enclosed in quotes. The font families in the last two examples, "serif" and "monospace", are so-called generic family names that specify a certain style of font. Other generic names include "sans-serif", "cursive", and "fantasy". The characters in a serif font have short lines as decorations such as at the top and bottom of an upper case "I". A "sans-serif" font lacks these decorations. The characters in a "monospace" font all have the same width. Monospace fonts are good for drawing characters that are meant to line up in columns.

Many other properties can be set using CSS, but I will not cover them here. I will use CSS only for borders, padding, background colors, and fonts.

Setting the style for many components can be tedious. A **CSS style sheet** can be used to apply style to all components of a given type as well as to individual components and sets of components. A style sheet is a file, usually with a name ending with .css. I will not discuss the syntax, but here is a style sheet that will apply some style rules to all *Labels* and *Buttons*:

```
Button {
   -fx-font: bold 16pt "Times New Roman";
   -fx-text-fill: darkblue;
}

Label {
   -fx-font: 15pt sans-serif;
   -fx-padding: 7px;
   -fx-border-color: darkred;
   -fx-border-width: 2px;
   -fx-text-fill: darkred;
   -fx-background-color: pink;
}
```

A style sheet file, just like an image file, can be a resource for a program. That is, it can be stored in the same place as the .class files for the program. Suppose that a style sheet named "mystyle.css" is in the program's top-level directory. You can then apply the style to all components in a scene with the statement

```
scene.getStylesheets().add("mystyle.css");
```

A *Scene* can have several style sheets, and style sheets can also be added to individual containers.

Section 6.3

Basic 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 target 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, which modifier keys (such as the shift key) are being held down, and which button on the mouse was pressed. When the user presses a key on the keyboard, on the other hand, it is a *KeyEvent* object that is created, containing information relevant to an event generated by using the keyboard. After the event object is constructed, it can be passed as a parameter to a designated method. That method is called an **event handler** for the event. In JavaFX, event handlers are often written as lambda expressions. By writing an event handler, 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 method 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 you didn't write it, there is a routine running somewhere that executes a loop of the form

```
while the program is still running:
    Wait for the next event to occur
    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 and key events in Java, and we'll cover the framework for handling events in general. We will also see how to make an animation.

6.3.1 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 **event listener**, which contains an event handler method that can respond to the event. An event listener is defined by an interface that specifies the event handling methods that it contains. Listeners for different kinds of events are defined by different interfaces. In most cases, the interface is a functional interface, defining a single event handler method; in that case, the listener can be given by a lambda expression.

For many kinds of events in JavaFX, listeners are defined by a functional interface named *EventHandler*, which defines the method `handle(event)`. The parameter to this method, `event`, is the event object that contains information about the event. When you provide a definition for `handle()`, you write the code that will be executed to handle the event.

(*EventHandler* is actually a parameterized type, something that we have not encountered before and will not encounter officially until Section 7.3.

Basically, all this really means is that *EventHandler* really defines many different types, with names like *EventHandler<MouseEvent>*, *EventHandler<KeyEvent>*, and *EventHandler<ActionEvent>*. The type *EventHandler<MouseEvent>* defines a `handle(event)` method in which the `event` is of type *MouseEvent*, the type *EventHandler<KeyEvent>* defines a `handle(event)` method in which the `event` is of type *KeyEvent*, and so on. Fortunately, you don't need to understand parameterized types in this chapter; you only need to know that the event object that you use when handling an event will have the appropriate type for that event. For example, when handling a mouse event, the event object is of type *MouseEvent*.)

Many events in JavaFX are associated with GUI components. For example, when the user presses a button on the mouse, the associated component is the one that contains the mouse cursor when the button is pressed. This object is called the **target** of the event. In order to respond to the event, you need to register a listener either with the target of the event or with some other object that knows about the event. For example, lets look again at this statement from HelloWorldFX.java, our first GUI program from Section 6.1:

```
helloButton.setOnAction( evt -> message.setText("Hello
World!") );
```

Here, `helloButton` is an object of type *Button*. When the user clicks on the button, an event of type *ActionEvent* is generated. The target of that event is `helloButton`. The method `helloButton.setOnAction()` registers an event listener that will receive notification of any *ActionEvents* from the button. The listener in this case is defined by a lambda expression. In the lambda expression, the parameter, `evt`, is the *ActionEvent* object, and the code in the lambda expression is what happens in response to the event. Most event handling in this chapter will be set up in a similar way.

For key events and some mouse events, it's not just the event target that gets a chance to respond to the event. For example, suppose that you press a mouse button over a *Canvas* that is inside a *BorderPane* that is in turn inside a *Scene*. The target of the mouse event is the *Canvas* but the *BorderPane* and the *Scene* also have a chance to respond to the event. That is, you can register a mouse event listener on any or all of these objects to respond to the event. The object that the listener is registered with is called the **source** of the

event. The event object parameter, `evt`, in an event handler method has both a source, given by `evt.getSource()`, and a target, given by `evt.getTarget()`; often they are the same, but they don't have to be. Note that the same event can be sent to several handlers. A handler can "consume" an event, by calling `evt.consume()`, to stop it from being sent to any additional handlers. For example, when you are typing in a text input box, the input box consumes the key events that you generate by typing, so that the scene doesn't get a chance to handle them.

(Actually, it's more complicated than that. For key events and some kinds of mouse events, the event first travels down through the scene and then through scene graph nodes that contain the event target; this is called the "event filtering" or "bubble down" phase of event processing. After reaching the target, the event travels back up through the scene graph and finally to the scene; this is the "event handling" or "bubble up" phase. The event can be consumed at any point along the way, and if that happens, the process stops. None of this is used in this chapter, but for more information, see the documentation for the `addEventFilter()` and `addEventHandler()` methods in the *Scene* and *Node* classes.)

Most of this section is concerned with mouse and key events. It is important to understand that many GUI programs do not need to deal with such events directly. Instead, you work with GUI components that are already programmed to handle mouse and key events on their own. For example, when the user clicks a *Button*, it is the button that listens for mouse events and responds to them. When the button detects that it has been clicked, it generates an *ActionEvent*. When you write an application that uses buttons, you program responses to *ActionEvents*, not to mouse events. Similarly, when the user types in a text input box, it is the input box that listens for key events and responds to them. Nevertheless, at base, it's mouse and keyboard events that drive the action in a program. It's useful to understand them—and you can do some interesting things by processing them directly.

6.3.2 Mouse Events

A mouse event is represented by an object of type *MouseEvent*. (In fact, mouse events can actually be generated by other input devices, such as a trackpad or touch screen; events from these devices are translated by the system into *MouseEvents*.) That class, and all of the classes related to mouse and key events, can be found in package `javafx.scene.input`. As the user manipulates the mouse, several kinds of event are generated. For example, clicking a mouse button generates three events, a "mouse pressed" event, a "mouse released" event, and a "mouse clicked" event. Simply moving the mouse generates a sequence of events as the mouse cursor moves from point to point on the screen. To respond to mouse events on a component, you can register listeners with that component. You can register a separate listener for each kind of mouse event on a component `c` using instance methods such as `c.setOnMousePressed(handler)` and `c.setOnMouseMoved(handler)`. The parameter is a mouse event handler, generally given as a lambda expression. Suppose, for example, that `canvas` is a component of type *canvas*, and that you would like a method, `redraw()`, to be called when the user clicks the canvas. You can make that happen by saying

```
canvas.setOnMousePressed( evt -> redraw() );
```

Generally, you would put this statement in the `start()` method of an *Application*, while setting up the GUI for the program. Mouse clicks on the canvas could be also be handled by the scene or by any scene graph node that contains the canvas, directly or indirectly, but it is much more usual for the target of a mouse event to handle the event.

Mouse event types include: `MouseEntered`, generated when the mouse cursor moves from outside a component into the component; `MouseExited`, generated when the mouse cursor moves out of a component; `MousePressed`, generated when the user presses one of the buttons on the mouse; `MouseReleased`, generated when the user releases one of the buttons on the mouse; `MouseClicked`, generated after a mouse released event if the user pressed and released the mouse button on the same component; `MouseDragged`, generated when the user moves the mouse while holding down a mouse button; and `MouseMoved`, generated when the user moves the mouse without holding down a button.

The target of a `MouseDragged`, `MouseReleased`, or `MouseClicked` event is the same component where the mouse button was pressed, even if the mouse has moved outside of that component. The target of a `MousePressed` or `MouseMoved` event is the component that contains the mouse cursor when the event occurs. And for `MouseEntered` and `MouseExited`, the target is the component that is being entered or exited.

Often, when a mouse event occurs, you want to know the location of the mouse cursor. This information is available from the *MouseEvent* parameter in 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 values of type double that 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 source of the event, where the top left corner of the component is (0,0). (The source is the component on which the event listener is registered; this is not necessarily the same as the event target, but it usually is.)

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). Not every computer has a Meta key. 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. For events triggered by a mouse button, you can determine which button was pressed or released by calling `evt.getButton()`, which returns one of the enumerated type constants `MouseButton.PRIMARY`, `MouseButton.MIDDLE`, or `MouseButton.SECONDARY`. Generally, the left mouse button is the primary button and the right mouse button is secondary. For events such as mouseEntered and mouseExited that are not triggered by buttons, `evt.getButton()` returns `MouseButton.NONE`.

The user can hold down several mouse buttons at the same time. If you want to know which mouse buttons are actually down at the time of an event, you can use the boolean-valued functions `evt.isPrimaryButtonDown()`, `evt.isMiddleButtonDown()`, and `evt.isSecondaryButtonDown()`.

As a simple example, suppose that when the user clicks a *Canvas*, `canvas`, you would like to draw a red rectangle at the point where the user clicked. But if the shift key is down, you want to draw a blue oval instead. An event handler to do that can be defined as:

```
canvas.setOnMousePressed( evt -> {
    GraphicsContext g = canvas.getGraphicsContext2D();
    if ( evt.isShiftDown() ) {
        g.setFill( Color.BLUE );
        g.fillOval( evt.getX() - 30, evt.getY() - 15, 60,
30 )
    }
    else {
        g.setFill( Color.RED );
        g.fillRect( evt.getX() - 30, evt.getY() - 15, 60,
30 );
    }
} );
```

To get a better idea of how mouse events work, you should try the sample program SimpleTrackMouse.java. This program responds 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 and buttons that are down. You can experiment with the program to see what happens as you do various things with the mouse. I also encourage you to read the source code.

6.3.3 Dragging

A drag gesture occurs when the user moves the mouse while holding down one of the buttons on the mouse. It is interesting to look at what a program needs to do in order to respond to dragging operations. The drag 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 event handlers, one for `MousePressed`, one for `MouseDragged`, and one for `MouseReleased`! Furthermore, the `MouseDragged` handler 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 double. It can also be useful to save the starting coordinates, where the original `MousePressed` event occurred, in instance variables. And I 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. Also, if the user presses a second mouse button without releasing the first, there will be two `MousePressed` events before the `MouseReleased` event; usually, you don't want the second `MousePressed` to start a new drag operation. The event-handling methods can use the value of `dragging` to check whether a drag operation is actually in progress. Often, I will write instance methods to handle the events, which in outline look something like this:

```
private double startX, startY; // Point where original
mouse press occurred.
private double prevX, prevY;   // Most recently processed
mouse coords.
private boolean dragging;      // Set to true when
dragging is in progress.
. . . // other instance variables for use in dragging

public void mousePressed(MouseEvent evt) {
   if (dragging) {
         // The user pressed a second mouse button before
releasing the first.
         // Ignore the second button press.
       return;
   }
   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.
     .
}
```

I will then install event handlers on the relevant component that simply call these methods:

```
c.setOnMousePressed( e -> mousePressed(e) );
c.setOnMouseDragged( e -> mouseDragged(e) );
c.setOnMouseReleased( e -> mouseReleased(e) );
```

Note that the event handlers in these statements simply call another method in the same class, and that method has the same parameter as the event handler. That means that it's possible to write the lambda expressions as method references (Subsection 4.5.4). The methods that are called are instance methods in the object "`this`", so the method references would have names like `this::mousePressed`, and the event handlers could be installed using

```
c.setOnMousePressed( this::mousePressed );
c.setOnMouseDragged( this::mouseDragged );
c.setOnMouseReleased( this::mouseReleased );
```

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. Here is a picture of the program window after some drawing has been done:

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.

In this program, all drawing is done on a single canvas that fills the entire window. The program is designed to work for any reasonable canvas size, that is, unless the canvas is too small. This means that coordinates are computed in terms of the actual width and height of the canvas. (The width and height are obtained by calling `canvas.getWidth()` and `canvas.getHeight()`.) This makes things quite a bit harder than they would be if we assumed some particular fixed size for the canvas. 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 canvas and the color palette along the right edge of the canvas. The gray border is 3 pixels wide. The colored rectangles are 50 pixels wide. Together with the 3-pixel border around the canvas 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 canvas.

A white square labeled "`CLEAR`" occupies the region beneath the colored rectangles on the right edge of the canvas. Allowing for this region, 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 `colorSpacing - 3`. The top of the `N`-th rectangle is located `(N*colorSpacing + 3)` pixels down from the top of the canvas, 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 canvas. 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 a `clearAndDrawPalette()` method that redraws the entire canvas. 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 `colorSpacing`. 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 canvas. This is not automatic, since as far as the computer is concerned, the border and the color bar are part of the canvas. If the user drags the mouse outside the white drawing area while drawing a curve, the `mouseDragged()` routine changes the `x` and `y` coordinates to make them lie within the drawing area.

6.3.4 Key Events

In Java, user actions become events in a program, with a GUI component as the target of the event. When the user presses a button on the mouse, the component that contains the mouse cursor is the target of the event. But what about keyboard events? When the user presses a key, what component is the target of the *KeyEvent* that is generated?

A GUI uses the idea of **input focus** to determine the target of keyboard events. At any given time, just one interface element on the screen can have the input focus, and that is where keyboard events are directed. If the interface element happens to be a JavaFX component, then the information about the keyboard event becomes an object of type *KeyEvent*, and it is delivered to any key event handlers that are listening for `KeyEvents` associated with that component. Note that because of the way key events are processed, the *Scene* object in the window that contains the focused component also gets a chance to handle a key event. If there is no other focused component in the window, then the scene itself will be the target for key events. In my sample programs, I will usually add key event handlers to the scene object.

A program generally gives some visual feedback to the user about which component has the input focus. For example, if the component is a text-input box, the feedback is usually in the form of a blinking text cursor. Another possible visual clue is to draw a brightly colored border around the edge of a component when it has the input focus. You might see that on a button that has focus. When a button has focus, pressing the space bar is equivalent to clicking the button.

If `comp` is any component, and you would like it to have the input focus, you can call `comp.requestFocus()`. 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. This is handled automatically by the components involved, without any programming on your part. However, some components do not automatically request the input focus when the user clicks on them. That includes, for example, a *Canvas*. Such a component can still receive the input focus if its `requestFocus()` is called. However, you can't automatically move the focus to that component with the tab key. To enable that, you can call `comp.setFocusTraversable(true)`. And you can test whether a component is focused by calling `comp.isFocused()`.

The focused component is contained in—or sometimes is itself—a window. That window is said to be the "focused" or "active" window. It is usually the front window on the screen. In JavaFX a *Stage* object is a window. You can call `stage.requestFocus()` to request that the window be moved to the front of the screen and become the active window. And you can call `stage.isFocused()` to test whether the window is active.

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 some cases, such as the shift key, 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 MacOS 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 JavaFX, there are three types of key event: `KeyPressed`, which is generated when the user depresses any key on the keyboard; `KeyReleased`, which is generated when the user releases a key that had been pressed; and `KeyTyped`, which is generated when the user types a character, possibly using a series of key presses and key releases. 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 highlighting the cards when the Shift key is pressed and removing the highlight when the Shift key is released.

There is one more complication. When you hold down a key on the keyboard, that key might **auto-repeat**. This means that it will generate multiple `KeyPressed` events with just one `KeyReleased` at the end of the sequence. 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.

Each key on the keyboard has a code that identifies it. In JavaFX, key codes are represented by enumerated type constants from the enum *KeyCode*. When an event handler for a `KeyPressed` or `KeyReleased` event 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.getCode()`. For example, when the user presses the shift key, this function will return the value `KeyCode.SHIFT`. You can find all the codes in the documentation for *KeyCode*, but names for most keys are easy to guess. Letter keys have names like `KeyCode.A` and `KeyCode.Q`. The arrow keys are named `KeyCode.LEFT`, `KeyCode.RIGHT`, `KeyCode.UP`, and `KeyCode.DOWN`. The space bar is `KeyCode.SPACE`. And function keys have names like `KeyCode.F7`.

In the case of a `KeyTyped` event, you want to know which character was typed. This information can be obtained by calling the function `evt.getCharacter()`. This function returns a value of type String that contains the character that was typed.

As a first example, you can check out the sample program KeyboardEventDemo.java. This program draws a small square on a canvas. The user can move the square left, right, up, and down by pressing arrow keys. This is implemented in a method

```
private void keyPressed( KeyEvent evt )
```

that is called by an event handler for `KeyPressed` events. The handler is installed on the *Scene* object in the `start()` method with the statement

```
scene.setOnKeyPressed( e -> keyPressed(e) );
```

In the `keyPressed()` method, the value of `evt.getCode()` is checked. If it's one of the arrow keys that was pressed, the canvas is redrawn to show the square in a different position.

The program also installs handlers for `KeyReleased` and `KeyTyped` events in a similar way. To give the `KeyTyped` handler something to do, it changes the color of the square when the user types "r", "g", "b", or "k". I encourage you to run the program and to read the entire source code.

6.3.5 AnimationTimer

There is another kind of basic event that I would like to introduce before turning to a more interesting example; that is, events that are used to drive an animation. The events in this case happen in the background, and you don't have to register a listener to respond to them. However, you do need to write a method that will be called by the system when the events occur.

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. In JavaFX, you can program an animation using an object of type *AnimationTimer* from package `javafx.animation`. An *AnimationTimer*, `animator`, has a method `animator.start()` that you can call to start the animation running or to restart it if it has been paused. It has the method `animator.stop()` to pause the animation. It also has a method `handle(time)`, but `handle()` is not a method that you call; it's one that you need to write to say what happens in the animation. The system will call your `handle()` method once for each frame of the animation. Its job is to do whatever is necessary to implement each frame.

The `handle()` method will be called on the JavaFX application thread, which means that you can do things like draw on a canvas or manipulate a GUI component. However, whatever you do should not take very long, since JavaFX animations are meant to run at at least 60 frames per second, which means `handle()` should ideally take less than 1/60 second to run.

AnimationTimer itself is an abstract class, and `handle()` is an abstract method. This means that to make an animation, you need to write a subclass of *AnimationTimer* and provide a definition for the `handle()` method. Suppose, for example, that you simply want to call a `draw()` method for each frame. This could be done as follows, using an anonymous subclass of *AnimationTimer* (see Subsection 5.8.3):

```
AnimationTimer animator = new AnimationTimer() {
    public void handle( long time ) {
        draw();
    }
};
```

Then, to get the animation started, you would need to call `animator.start()`. This could all be done in an application's `start()` method.

The parameter, `time`, gives the current time, measured as the number of nanoseconds since some arbitrary time in the past (the same arbitrary time that is used by `System.nanoTime()`). You can use `time` in the calculations that you do for the frame, as a way of making each frame different from the next. Another option is to use a frame number that increases by one each time a frame is drawn, but you should keep in mind that the rate at which `handle()` is called can vary, and the time between frames is not guaranteed to be 1/60 second.

6.3.6 State Machines

We are ready to look at a program that uses animation and key events to implement a simple game. The program uses an *AnimationTimer* to drive the animation, and it uses a number of instance variables to keep track of the current state of the game. The idea of "state" is an important one.

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 depends on what state it's in when the event occurs. 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 state be taken into account when drawing the content of a canvas? 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 *KeyboardEventDemo* program, discussed above, the state of the program is recorded in instance variables such as `squareColor`, `squareLeft`, and `squareTop`, which record the color and position of the colored square. These state variables are used in a `draw()` method that draws the square on a canvas. Their values are changed in the 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 program is defined in the source code file SubKiller.java. As usual, it would be a good idea to compile and run the program as well as read the full source code. Here is a picture:

The entire application window is filled by a canvas. The program shows a black "submarine" near the bottom of the canvas. The submarine moves erratically back and forth near the bottom of the window. 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 application 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` 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 those objects have instance 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 an *AnimationTimer*. Each time the animator's `handle()` method is called, some of the state variables have to change to get ready to draw next frame of the animation. The changes are made in the `handle()` method before redrawing the canvas. 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 `handle()` method calls these methods with the statements

```
boat.updateForNewFrame();
bomb.updateForNewFrame();
sub.updateForNewFrame();
```

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 by one in each frame, and if it has reached 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 following lines to change the value of `isMovingLeft` at random times:

```
if ( Math.random() < 0.02 ) {
    isMovingLeft = ! isMovingLeft;
}
```

There is a 1 in 50 chance that `Math.random()` will be less than 0.02, so the statement "`isMovingLeft = ! isMovingLeft`" is executed in one out of every fifty 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 handler for `KeyPressed` events. 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. The handler is a long lambda expression that is registered with the `scene` in the application's `start()` method:

```
scene.setOnKeyPressed( evt -> {
           // The key listener responds to keyPressed
events on the canvas.
           // The left- and right-arrow keys move the boat
while down-arrow
           // releases the bomb.
       KeyCode code = evt.getCode();  // Which key was
pressed?
       if (code == KeyCode.LEFT) {
           boat.centerX -= 15;
       }
       else if (code == KeyCode.RIGHT) {
           boat.centerX += 15;
       }
       else if (code == KeyCode.DOWN) {
           if ( bomb.isFalling == false )
               bomb.isFalling = true;
       }
   } );
```

Note that it's not necessary to redraw the canvas in this method, since this canvas 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 the key event handler, but I chose to check for this in another routine, in the boat object, since it seems natural to let the boat object be responsible for keeping itself on the screen.

I encourage you to read the source code in SubKiller.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.

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.

6.3.7 Observable Values

There is one more kind of basic event that plays an important role in JavaFX: events that are generated when an **observable value** is modified. There is an example in the `SubKiller` program. A *Stage*, `stage`, has a property of type *ObservableBooleanValue* that tells whether or not `stage` is currently the focused window. You can access the property by calling `stage.focusedProperty()`. When the value of an *ObservableBooleanProperty* changes, an event is generated. You can register a *ChangeListener* with the property, containing an event handler method that will be called when the event occurs. The handler method in this case has three parameters: the observable property that generated the event, the previous value of the property, and the new value. For an *ObservableBooleanValue*, the old and new values are of type boolean. There are other observable value types, such as *ObservableIntegerValue*, *ObservableStringValue*, and *ObservableObjectValue*.

When I first wrote `SubKiller`, the animation would continue to run even when the `SubKiller` window was not the focused window, which I found annoying when I was trying to work in another window. I decided to pause the animation when the window loses the input focus and to restart it when the window regains focus. When the window loses or gains focus, the value of the observable boolean property `stage.focusedProperty()` changes. To react to that change, I added a change listener to the property, which stops the animation when the value of the property changes to `false` and starts the animation when the value changes to `true`. So, I added this code to the `start()` method:

```
stage.focusedProperty().addListener( (obj,oldVal,newVal) -
> {
            // This listener turns the animation off when
this program's
            // window does not have the input focus.
        if (newVal) { // The window has gained focus.
            timer.start();
        }
```

```
        else {  // The window has lost focus.
            timer.stop();
        }
        draw(); // Redraw canvas. (Appearance changes
depending on focus.)
    });
```

The `addListener()` method of an observable property registers a change listener with the property. Note that the lambda expression for the event handler takes three parameters. The only one that I need here is `newVal` which represents the current value of the stage's `focused` property. The `draw()` method draws draws some things differently, depending on whether the stage is focused. It tests for that calling `stage.isFocused()`.

JavaFX GUI components have many observable properties, of various types. For example, the text on a *Button* is a property of type *ObservableStringProperty*, and the width and the height of a canvas are values of type *ObservableDoubleProperty*. We will encounter more examples in the next section.

Section 6.4

Basic Controls

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 button, it changes appearance when 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, and releasing the mouse at that time will not trigger the button. 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 `javafx.scene.control.Button`. A

Button object draws itself and processes mouse, keyboard, and focus events on its own. You only hear from the *Button* 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 *Button* object creates an event object belonging to the class `javafx.event.ActionEvent`. The event object is sent to any registered listener to tell it that the button has been pushed. Your program gets only the information it needs—the fact that a button was pushed.

Many standard components that are defined as part of the JavaFX graphical user interface API are defined by subclasses of the class *Control*, from package `javafx.scene.control`. Controls (with just a couple of exceptions) can be manipulated by the user to generate input and events for the program. A number of useful methods are defined for controls. I begin by mentioning three methods that can be used with any *Control* `control`:

- `control.setDisable(true)` can be called to disable the control. The control can be re-enabled with `control.setDisable(false)`. When a control is disabled, its appearance might change, and it cannot be the target of mouse or key events. This function can actually be called with any scene graph node, not just controls; when you disable a node, any nodes contained inside that node are also disabled. There is a boolean-valued function, `control.isDisabled()` that you can call to discover whether the control is disabled, either because it was explicitly disabled or because it is inside a container node that was explicitly disabled.
- `control.setTooltip(new Tooltip(string))` sets the specified string as a "tool tip" for the control. The tool tip is displayed if the mouse cursor is inside the control and the mouse is not moved for a few seconds. The tool tip should give some information about the meaning of the control or how to use it.
- `control.setStyle(cssString)` sets the CSS style of the control. CSS was discussed in Subsection 6.2.5.

Note that using a control, or any scene graph node, 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 control components that are available in JavaFX. They are all defined by classes in the package *javafx.scene.control*. In the next section we will consider the problem of laying out components in containers.

6.4.1 ImageView

But before we turn to controls, I want to mention one other node type: *ImageView* from package `javafx.scene.image`. Recall from Subsection 6.2.3 that an *Image* object represents a picture, and that images can be loaded from resource files. An *Image* can be drawn on a *Canvas*, but an *Image* is not a *Node*. That is, it can't be part of a scene graph.

An *ImageView* is a scene graph node that is a simple wrapper for an image. Its purpose is simply to display the image. It makes it possible to add the image to a scene graph without having to draw the image on a canvas. The image can be specified as a parameter to the *ImageView* constructor. For example, suppose that "icons/tux.png" is the path to an image resource file. Then an *ImageView* can be created to display the image like this:

```
Image tux = new Image("icons/tux.png");
ImageView tuxIcon = new ImageView( tux );
```

In this case, I am thinking of the image as an "icon," that is, a small image that is typically used on a button, label, or menu item to add a graphical element to the usual plain text. In fact, we'll see that you can do exactly that in JavaFX.

6.4.2 Label and Button

The first four controls that we will look at have something in common: They all display a string of text to the user, which the user can view but not edit. Furthermore they can all display a graphical element in addition to or instead of the text. The graphic can be any *Node* but is usually a small icon,

implemented as an object of type *ImageView*. In fact, all four types of controls inherit their behavior from a common superclass named *Labeled*. In Subsection 6.6.2, we look at menu items, which also inherit from *Labeled*. That class defines a number of instance methods that can be used with labels, buttons, and other labeled controls. Those methods include:

- `setText(string)` for setting the text that is displayed on the control. The text can be multi-line. The new line character, "\n", in the `string` represents a line break.
- `setGraphic(node)` for setting the control's graphical element.
- `setFont(font)` for setting the font that is used for the text.
- `setTextFill(color)` for setting the paint that is used for drawing the text.
- `setGraphicTextGap(size)` for setting the amount of space that is put between the text and the graphic. The parameter is of type double.
- `setContentDisplay(displayCode)` for setting where the graphic should be placed with respect to the text. The parameter is a constant from an enum, *ContentDisplay*, such as `ContentDisplay.LEFT`, `ContentDisplay.RIGHT`, `ContentDisplay.TOP` or `ContentDisplay.BOTTOM`.

All of these setter methods have corresponding getters, such as `getText()` and `getFont()`. I have not listed a setter method for the background color. It is possible to set a background color, but it's cumbersome. (Setting the background color of a control, `c`, to white looks like this:

```
c.setBackground(new Background(new
BackgroundFill(Color.WHITE,null,null)));
```

where *Background* and *BackgroundFill* are in package `javafx.scene.layout`.) It is more easily done by setting the CSS style of the control with the `setStyle()` method. CSS is also useful for setting the border and for setting padding (the amount of empty space left around the content).

A *Label* is certainly the simplest type of control. It adds almost nothing to the *Labeled* class. Its purpose is simply to display some unedited text and/or a

graphic to the user. The label class has two constructors. The first has one parameter of type *String* specifying the text for the label. The second adds a parameter of type *Node* specifying a graphic for the label. For example, assuming that `tuxIcon` is the *ImageView* object from the previous subsection,

```
Label message = new Label("Hello World");
Label linuxAd = new Label("Choose Linux First!", tuxIcon);
```

The default background of a label is completely transparent, and the label has no border or padding by default. Often I would like to add at least some padding. Here is an example of setting all three properties using CSS:

```
message.setStyle("-fx-border-color: blue; -fx-border-
width: 2px; " +
                              "-fx-background-color: white; -
fx-padding: 6px");
```

You've already seen buttons used in Section 6.1. A *Button*, like a *Label*, displays some text and/or a graphic to the user, and the *Button* class, like the *Label* class, has two constructors:

```
Button stopButton = new Button("Stop");
Button linuxButton = new Button("Get Linux", tuxIcon);
```

When the user clicks a button, an event of type *ActionEvent* is generated. You can register an event handler for the action with the button's `setOnAction` method. For example,

```
stopButton.setOnAction( e -> animator.stop() );
```

In addition to the methods inherited from *Labeled*, a button has the instance methods `setDisable(boolean)` and `setToolTip(string)` that were mentioned at the beginning of this section. The `setDisable()` and `setText()` methods 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!" For example, suppose that we want to use two buttons to start and stop an *AnimationTimer*, `animator`. When the animation is running, the

start button should be disabled, and when the animation is paused, the stop button should be disabled. The code for this might look like:

```
Button startButton = new Button("Run Animation");
Button stopButton = new Button("Stop Animation");
stopButton.setDisable(true); // Stop button is initially
disabled.
startButton.setOnAction( e -> {
    animator.start();
    startButton.setDisable(true);
    stopButton.setDisable(false);
} );
stopButton.setOnAction( e -> {
    animator.stop();
    startButton.setDisable(false);
    stopButton.setDisable(true);
} );
```

This ensures that the user can't try to start the animation when it is already started or stop it when it is already stopped.

Often, there is a button that triggers some default action in a program. For example, the user might enter data into some text input boxes and then click a "Compute" button to process the data. It would be nice if the user could just press Return when done typing, instead of having to click the button. In JavaFX, you can make a *Button*, `button`, into the default button for the window that contains it by calling

```
button.setDefaultButton(true);
```

When a window has a default button, then pressing the Return (or Enter) key on the keyboard is equivalent to clicking the default button, unless the key event generated by the Return key is consumed by another component. This can be very convenient for the user.

6.4.3 CheckBox and RadioButton

A *CheckBox* is a component that has two states: selected or unselected. (Being "selected" means that the checkbox is checked.) 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:

```
CheckBox showTime = new CheckBox("Show Current Time");
```

CheckBox is a subclass of *Labeled*, so a checkbox can also have a graphic, and all the instance methods from the *Labeled* class can also be used with checkboxes. (There is no constructor that specifies a graphic for the checkbox; any graphic has to be set by calling `setGraphic`(node).)

Usually, it's the user who sets the state of a *CheckBox* by clicking on it, but you can also set the state programmatically. 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 a handler with the checkbox by calling its `setOnAction()` method. (Note that if you change the state by calling the `setSelected()` method, no *ActionEvent* is generated. However, there is another method in the *CheckBox* class, `fire()`, which simulates a user click on the checkbox and does generate an *ActionEvent*.)

It is actually possible for a checkbox to be in a third state, called "indeterminate," although that possibility is turned off by default. See the API documentation for details.

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, where at most one radio button in a given group can be selected at any given time. Radio button groups let the user select one choice among several alternatives. In JavaFX, a radio button is represented by an object of type *RadioButton*. When used in isolation, a *RadioButton* acts just like a *CheckBox*, and it has the same constructor, methods, and events, including methods inherited from *Labeled*. Ordinarily, however, a *RadioButton* is used in a group. A group of radio buttons is represented by an object belonging to the class *ToggleGroup*. A *ToggleGroup* is **not** a component and does not itself have a visible representation on the screen. A *ToggleGroup* 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 *RadioButton* object for each button in the group, and you must create one object of type *ToggleGroup* to organize the individual buttons into a group. Each *RadioButton* must be added individually to the scene graph, so that it will appear on the screen. (A *ToggleGroup* plays no role in the placement of the buttons on the screen.) Each *RadioButton* must also be added to the *ToggleGroup*. You do that by calling the radio button's `setToggleGroup(group)` instance method. 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 group of radio buttons that can be used to select a color:

```
RadioButton redRadio, blueRadio, greenRadio, yellowRadio;
         // Variables to represent the radio buttons.
         // These might be instance variables, so that
         // they can be used throughout the program.

ToggleGroup colorGroup = new ToggleGroup();

redRadio = new RadioButton("Red");   // Create a button.
redRadio.setToggleGroup(colorGroup); // Add it to the
ToggleGroup.
```

```
blueRadio = new RadioButton("Blue");
blueRadio.setToggleGroup(colorGroup);

greenRadio = new RadioButton("Green");
greenRadio.setToggleGroup(colorGroup);

yellowRadio = new RadioButton("Yellow");
yellowRadio.setToggleGroup(colorGroup);

redRadio.setSelected(true);  // Make an initial selection.
```

As an alternative to calling `redRadio.setSelected(true)`, you can use the `selectToggle()` instance method in the *ToggleGroup* class to select the radio button:

```
colorGroup.selectToggle( redRadio );
```

Just as for checkboxes, it is not always necessary to register listeners for radio buttons. You can test the state of an individual *RadioButton* by calling its `isSelected()` method, or you can call the toggle group's `getSelectedToggle()` method. The return type of this method is *Toggle*, which is an interface implemented by *RadioButton*. For example:

```
Toggle selection = colorGroup.getSelectedToggle();
if (selection == redRadio) {
    color = Color.RED;
}
else if (selection == greenRadio){
    .
    .
    .
```

Here's what these radio buttons look like, lined up vertically in a container:

6.4.4 TextField and TextArea

The *TextField* and *TextArea* classes represent components that contain text that can be edited by the user. A *TextField* holds a single line of text, while a *TextArea* can hold multiple lines. It is also possible to set a *TextField* or *TextArea* 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, *TextInputControl*, which defines their common properties.

TextField and *TextArea* have many methods in common. The instance method `setText(text)`, 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*. You can add a *String* of text onto the end of the text that is already in the component by calling the instance method `appendText(text)`. The text in the `setText()` and `appendText()` methods can include "\n" characters to represent line breaks; in a *TextField* they will be ignored. The instance method `setFont(font)` can be used to change the font that is used in the text component.

If you want to stop the user from modifying the text, you can call `setEditable(false)`. Call the same method with parameter `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.

A substring of the text in a text component can be "selected." The selected text is highlighted and can be cut or copied from the text component. (The user can right-click in the component to bring up a pop-up menu of editing commands.) *TextInputComponent* has several instance methods for working with the text selection, but I only mention one of them: `selectAll()`, which selects the entire string of text in the text component.

For example, when I discover an error in the user's input in a *TextField*, `input`, I usually call both `input.requestFocus()` and `input.selectAll()`. This helps the user see where the error occurred and lets the user start typing the correction immediately. If the user starts typing,

the old text in the input box, which is highlighted, will automatically be deleted.

Both the *TextField* class and the *TextArea* class have two constructors. One of the constructors has no parameter and makes an initially empty text input box. The second constructor takes a parameter of type *String*, specifying the initial content of the box.

A *TextField* has a preferred number of columns, which by default is 12. This determines the preferred width of the text field, that is, the size that is used if the size is not reset by the program or when the text field is laid out by a container. The preferred number of columns for a *TextField*, `input`, can be changed by calling `input.setPrefColumnCount(n)`, where `n` is a positive integer.

Similarly, a *TextArea* has both a preferred number of columns and a preferred number of rows, which by default are 40 and 10. The value can be changed using the *TextArea* instance methods `setPrefColumnCount(n)` and `setPrefRowCount(n)`.

The *TextArea* class adds a few useful methods to those inherited from *TextInputControl*, including some methods for getting and setting the amount by which the text has been scrolled. Most important is `setWrapText(wrap)`, where `wrap` is of type boolean. This method says 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, with the line break occurring between words if possible; 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, a *TextArea* comes with scroll bars. The scroll bars are visible only when they are needed because the text cannot fit in the available space.

See the sample program TextInputDemo.java for a short example of using a text field and a text area. Here is the window from that program, after the text has been edited and scrolled down a few lines:

(I pasted the source code of the program into the text area!)

6.4.5 Slider

A *Slider* 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, or by clicking at some point along the bar. A slider can, optionally, be decorated with tick marks and with labels. This picture, from the sample program SliderDemo.java, shows three sliders with different decorations and with different ranges of values:

Here, the second slider is decorated with tick marks, 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 *Sliders* 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 Slider(double minimum, double maximum, double
value)
```

If the parameters are omitted, the values 0, 100, and 0 are used. By default, a slider is horizontal, but you can make it vertical by calling its method `setOrientation(Orientation.VERTICAL)`. *Orientation* is an enum from package `javafx.geometry`.

The current value of a *Slider* can be read at any time with its `getValue()` method, which returns a value of type double. If you want to change the value programmatically, you can do so with the method `setValue(val)`, which takes a parameter of type double. If the specified value is not between the minimum and maximum slider values, it is adjusted to lie in that range.

If you want to respond immediately when the user changes the value of a slider, you can register a listener with the slider. *Sliders*, unlike other components we have seen, do not generate `ActionEvents`. Instead, they have an observable property of type `Double` that represents the value. (See Subsection 6.3.7.) You can access the value property of a *Slider*, `slider`, by calling `slider.valueProperty()`, and you can add a listener to the property that will be called whenever the value changes. For example, in the sample program, I add listeners to the sliders with commands such as

```
slider1.valueProperty().addListener( e ->
sliderValueChanged(slider1) );
```

The listener will be called whenever the value changes, either because the user is dragging the knob on the slider or because the program calls `setValue()`. If you want to know whether the user generated the event by dragging the slider's knob, call the slider's boolean-valued `isValueChanging()` method, which returns `true` if the user is dragging the knob.

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 `setMajorTickSpacing(x)` indicates that there should be a major tick mark every `x` units along the slider. The parameter is of type double. (The spacing is in terms of values on the slider, not pixels.) To control the minor tick marks, you can call `setMinorTickCount(n)`. The parameter is an int that specifies how many minor tick marks are placed between consecutive major tick marks. The default value is 4. If you don't want minor tick marks, set the count to zero. Calling these methods is not enough to make the tick marks appear. You also have to call `setShowTickMarks(true)`. For example, the second slider in the sample program was created and configured using the commands:

```
slider2 = new Slider();  // Use default values (0,100,0)
slider2.setMajorTickUnit(25); // space between big tick
marks
slider2.setMinorTickCount(5); // 5 small tick marks
between big tick marks.
slider2.setShowTickMarks(true);
```

Labels on a slider are handled similarly. A label will be placed at every major tick mark, but some labels will be omitted if they would overlap. To see the labels, you need to call `setShowTickLabels(true)`. For example, the third slider in the above illustration was created and configured to show labels with the commands:

```
slider3 = new Slider(2000,2100,2022);
slider3.setMajorTickUnit(50); // tick marks are not shown!
slider3.setShowTickLabels(true)
```

The value of a slider is of type double. You might want to restrict the value to be an integer, or to be some multiple of a given value. What you can do is call `slider.setSnapToTicks(true)`. After the user finishes dragging the slider's knob, the value will be moved to the value at the nearest major or minor tick mark, even if the tick marks are not visible. The value will **not** be restricted while the user is dragging the knob; the value is just adjusted at the end of the drag. The value set by calling `setValue(x)` is also **not** restricted, but there is another method, `adjustValue(x)`, that will set the value to the

value at the tick mark nearest to `x`. For example, if you want a slider to snap to integral values in the range 0 to 10, you could say:

```
Slider sldr = new Slider(0,10,0);
sldr.setMajorTickUnit(1);  // major ticks 1 unit apart
sldr.setMinorTickCount(0); // no minor tick marks
sldr.setSnapToTicks(true);
```

In the sample program, the third slider has been configured to snap to an integer value at the end of a drag operation.

Section 6.5

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 placed 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. In `JavaFX` terms, a container is a scene graph node that can contain other scene graph nodes as "children." The children of a container must be "laid out," which means setting their sizes and positions. It is possible for you to program the layout yourself, but layout is usually done automatically by the container. Different kinds of containers implement different policies for laying out their child nodes. In this section, we will cover some common types of JavaFX containers and their layout policies. In this section and the next, we will look at several programming examples that use components and layout.

Because a container is itself a scene graph node, you can add a container as a child of another container. This makes complex nesting of components possible, as shown in this illustration:

In this picture, a large container holds two smaller containers. Each of the two smaller containers in turn holds three components.

Every scene graph node has a minimum width and height, a maximum width and height, and a preferred width and height. A container consults these values when it is deciding how to lay out its children. (Some nodes, however, are not resizable, meaning not meant to be resized by a container during layout; for such components, the minimum, maximum, and preferred width and height are effectively equal to the component's actual size. *Canvas* and *ImageView* are examples of non-resizable components.) In general, a container will compute its own preferred size based on the components that it contains. The container's preferred size will allow each component that it contains to have at least its preferred size. The container will compute its own minimum and maximum sizes in a similar way, based on the minimum and maximum sizes of its children.

When setting the size of a child while doing layout, most containers will **not** set the width of the child to be less than the child's minimum width or greater than the child's maximum width, and the same for the child's height. This will be true even if it means that the child will overlap other components or will extend outside the container. (The part outside the container might or might not be shown, depending on the container.) This can also leave empty space in the container.

Resizable nodes, which includes controls and most containers, have instance methods for setting the minimum, preferred, and maximim width: `setMinWidth(w)`, `setPrefWidth(w)`, and `setMaxWidth(w)`, where the parameter is of type double, with similar instance methods for the height. You can also set width and height values simultaneously with methods such as `setMaxSize(w,h)` and `setPrefSize(w,h)`. For a container, the values set by these methods will override the values that would have been computed by the container based the children that it contains.

In JavaFX, containers that do layout are defined by the class *Pane* and its subclasses. (*Pane* and its subclasses are in package `javafx.scene.layout`.) Here, we look at a few of these layout classes, starting with using *Pane* directly.

6.5.1 Do Your Own Layout

Sometimes, you would like to do your own layout, instead of letting it up to one of the standard container classes. You can get complete control of the size and location of a node by setting it to be "unmanaged":

```
node.setManaged(false);
```

When you do that, **any** container that contains `node` as a child will completely ignore `node`, as far as layout is concerned. It is your responsibility to set its location and (for resizable nodes) size. Note that an unmanaged node must still be placed inside a container for it to appear on the screen.

For our first layout example, we'll look at using the *Pane* class itself for the container. The *Pane* will manage the size, but not the location, of the managed nodes that it contains. That is, by using a *Pane* as container, you assume responsibility for setting the positions of the child nodes. The *Pane* will, by default, resize each child to its preferred size; if you want to take over that job too, you need to make the child unmanaged.

If `node` is any scene graph node that is unmanaged or is in a container that does not set the location of its child nodes, then you can set the location by calling

```
node.relocate( x, y );
```

This puts the top left corner of the node at the point `(x,y)`, measured in the coordinate system of the container that contains the node. Similarly, you can set the size of `node`, when that is allowed, by calling

```
node.resize( width, height )
```

The width and height are measured in pixels.

We will look at an example that contains four components: two buttons, a label, and a canvas that displays a checkerboard pattern:

The program is just an example of layout; it doesn't do anything, except that clicking the buttons changes the text of the label. (I will use this example in Section 7.6 as a starting point for a checkers game.)

This example uses a *Pane* as the root node of the scene and as the container for the four components. This means that the program is responsible for setting the locations of the components, by calling their `relocate()` method. (Without that, they will all appear with their top left corners at the default location, (0,0)!) After looking at the result, I decided that I wanted the two buttons to have the same size and to be larger than their preferred sizes. For that to work, the buttons had to be made unmanaged. (Otherwise, calling their `resize()` method would have no effect because the pane would set them back to their preferred size.) Here is the code from the application's `start()` method that creates the four components and configures them, including setting their locations and sizes:

```
/* Create the child nodes. */

board = new Checkerboard(); // a nested subclass of Canvas
board.draw();  // draws the content of the checkerboard

newGameButton = new Button("New Game");
newGameButton.setOnAction( e -> doNewGame() );
```

```
resignButton = new Button("Resign");
resignButton.setOnAction( e -> doResign() );

message = new Label("Click \"New Game\" to begin.");
message.setTextFill( Color.rgb(100,255,100) ); // Light
green.
message.setFont( Font.font(null, FontWeight.BOLD, 18) );

/* Set the location of each child by calling its
relocate() method */

board.relocate(20,20);
newGameButton.relocate(370, 120);
resignButton.relocate(370, 200);
message.relocate(20, 370);

/* Set the sizes of the buttons.  For this to have an
effect, make
 * the butons "unmanaged."  If they are managed, the Pane
will set
 * their sizes. */

resignButton.setManaged(false);
resignButton.resize(100,30);
newGameButton.setManaged(false);
newGameButton.resize(100,30);
```

The *Pane* that holds the components is the root node of the scene. The window for the program will be sized to be just large enough to hold the *Pane* at its preferred size. By default, a *Pane* computes its preferred size to be just large enough to show all of its managed children. Since I made the buttons unmanaged, they would not be included in the preferred size. To make the window large enough to include them (and to allow more empty space below the label), the program sets the preferred width and height of the pane:

```
Pane root = new Pane();
root.setPrefWidth(500);
root.setPrefHeight(420);
```

The buttons, label, and board must still be added as children of the pane, to make them appear on the screen. This is done with the command:

```
root.getChildren().addAll(board, newGameButton,
resignButton, message);
```

Alternatively, they could have been added one by one using statements such as

```
root.getChildren().add(board);
```

or the child nodes could have been given as parameters to the constructor:

```
Pane root = new Pane(board, newGameButton, resignButton,
message);
```

The pane still has to be made the root of the scene, the scene has to be placed in the stage, and the stage has to be configured and shown on the screen. See the source code, OwnLayoutDemo.java, for the complete story.

It's fairly easy in this case to get a reasonable layout. It's much more difficult to do your own layout if you want to allow for changes in the size of the window. To do that, you would probably write your own subclass of *Pane* (or of its superclass, *Region*), and override the `layoutChildren()` method, which is called by the system when a container needs to do layout, including when the container is resized. However, my real advice is that if you want to allow for changes in the container's size, try to find a standard container that will do the layout work for you.

6.5.2 BorderPane

A *BorderPane* is a subclass of *Pane* that is designed to display one large, central component, with up to four smaller components arranged around the edges of the central component. The five positions are referred to as center, top, right, bottom, and left. 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.

The *BorderPane* class has two constructors, one with no parameters and one that takes a single parameter giving the center child of the pane. Child nodes can be added to a *BorderPane*, `pane`, using the methods

```
pane.setCenter(node);
pane.setTop(node);
pane.setRight(node);
pane.setBottom(node);
pane.setLeft(node);
```

Calling one of these methods with parameter `null` will remove the child from that position in the pane.

A *BorderPane* sets the sizes of its child nodes as follows, except that the width and height of a component is always restricted to lie between the minimum and maximum width and height of that component: The top and bottom components (if present) are shown at their preferred heights, but their width is set equal to the full width of the container. The left and right components are shown at their preferred widths, but their height is set to the height of the container, minus the space occupied by the top and bottom components. Finally, the center component takes up any remaining space.

The default preferred size of *BorderPane* is set just big enough to accommodate the preferred sizes of its (managed) children. The minimum size is computed in a similar way. The default maximum size of a *BorderPane* is unlimited.

For some subclasses of *Pane*, it is possible to tweak the layout of the children by applying things called **layout constraints**. For example, what happens in a *BorderPane* if a child cannot be resized to exactly fit the available space? In that case, the child has a default position within the available space. The center child is centered within the center space of the pane, the bottom child is placed at the bottom left corner of the bottom space in the pane, and so on. You can change this default placement using a static method from the *BorderPane* class:

```
BorderPane.setAlignment( child, position );
```

where `child` is the child node whose position you want to tweak and `position` is one of the constants from the enumerated type *Pos*, from package `javafx.geometry`. Possible positions include `Pos.CENTER`, `POS.TOP_LEFT`, `Pos.BOTTOM_RIGHT`, and so on. (I find it strange that the alignment is set using a static method, but that's the way layout constraints work in JavaFX.)

You can also set a **margin** for any child of a *BorderPane*. A margin is empty space around the child. The background color of the pane will show in the margin. A margin is specified as a value of type *Insets*, from package `javafx.geometry`. An object of type *Insets* has four double properties, `top`, `right`, `bottom`, and `left`, which can be specified in the constructor:

```
new Insets(top,right,bottom,left)
```

There is also a constructor that takes a single parameter, which is used as the value for all four properties. When used as a margin for a child node, the properties specify the width of the margin along the top, right, bottom, and left edges of the child. The margin can be specified using another static method:

```
BorderPane.setMargin( child, insets );
```

For example,

```
BorderPane.setMargin( topchild, new Insets(2,5,2,5) );
```

Remember that you can also tweak the appearance of a container using CSS (Subsection 6.2.5), which is the easiest way to set a border or background color for the container.

6.5.3 HBox and VBox

It is common to want to lay out a set of components in a horizontal row or in a vertical column. For that, the classes *HBox* and *VBox* can be used. *HBox* is a subclass of *Pane* that lays out its children next to each other, in a horizontal row. *VBox* is a subclass of *Pane* that lays out its children in vertical column. An *HBox* might be used as the bottom component in a *BorderPane*, making it possible to line up several components along the bottom of the border pane. Similarly, a *VBox* might be used as the left or right component of a *BorderPane*. Here, I will only discuss *HBox*, but *VBox* is used in an entirely analogous way.

An *HBox*, `hbox`, can be configured to leave a gap between each child and the next. The amount of the gap is given by a value of type double, which can be set by calling

```
hbox.setSpacing( gapAmount );
```

The default value is zero. Children can be added to `hbox` in the same way as for *Pane*, that is, by calling `hbox.getChildren().add(child)` or `hbox.getChildren().addAll(child1,child2,...)`. The *HBox* class has a constructor with no parameters, as well as one that takes the size of the gap as its first parameter, optionally followed by child nodes to be added to the box.

By default, an *HBox* will resize its children to their preferred widths, possibly leaving some blank extra space on the right. (The blank space would be there because the width of the *HBox* has been set to be larger than its preferred width.) If using the children's preferred widths would make the total width greater than the actual width of the *HBox*, it will shrink the children, within the limit of their minimum widths. The height of the children will be set to the full available height in the *HBox*, but, as always, within the limits set by their minimum and maximum heights.

Perhaps you would like the children to grow beyond their preferred widths, to fill the available space in an *HBox*. To make that happen, you need to set a layout constraint on each child that you want to grow, using a static method:

```
HBox.setHgrow( child, priority );
```

The second parameter is a constant of the enumerated type *Priority*, from package `javafx.scene.layout`. The value will likely be `Priority.ALWAYS`, which means that the child will always get a share of the extra space. The child's width will still be limited by its maximum width, so you might need to increase that maximum to get the child to grow to the extent that you want.

As an example, suppose that an *HBox* contains three *Buttons*, `but1`, `but2`, and `but3`, and that you would like them to grow to fill the entire *HBox*. You need to set the `HGrow` constraint on each button. Furthermore, since the maximum width of a button is equal to its preferred width, you need to increase each button's maximum width. Here, I set the maximum width to be `Double.POSITIVE_INFINITY`, which means that the button can grow without any limit:

```
HBox.setHgrow(but1, Priority.ALWAYS);
HBox.setHgrow(but2, Priority.ALWAYS);
HBox.setHgrow(but3, Priority.ALWAYS);
but1.setMaxWidth(Double.POSITIVE_INFINITY);
but2.setMaxWidth(Double.POSITIVE_INFINITY);
but3.setMaxWidth(Double.POSITIVE_INFINITY);
```

Any extra space will be distributed equally to the three buttons and added on to their preferred widths. This does not mean that they will all have the same width, because their original, preferred widths might not be equal. For a sample program in the next section, I wanted the three buttons in an *HBox* to be the same size. To accomplish that, I simply gave all three buttons the same large preferred width:

```
but1.setPrefWidth(1000);
but2.setPrefWidth(1000);
but3.setPrefWidth(1000);
```

The *HBox* will then shrink the buttons proportionately to fit in the available space, and they all end up with the same width.

There are other ways to tweak the layout in an *HBox*. You can add a margin around any child using a static method similar to the one that is used for the same purpose in a *BorderPane*. You can set the *HBox* to use the preferred heights of its children, instead of stretching them to fit the height of the hbox by calling `hbox.setFillHeight(false)`. You can say where in the hbox the children are placed, when they do not fill the entire hbox by calling `hbox.setAlignment(position)`; the parameter is of type *Pos* with a default of `Pos.TOP_LEFT`. And, of course, you can apply CSS style properties.

As an example, we look at a GUI built entirely using *HBox* and *VBox*. The complete program can be found in SimpleCalc.java. The program has two *TextFields* where the user can enter numbers, four *Buttons* that the user can click to add, subtract, multiply, or divide the two numbers, and a *Label* that displays the result of the operation. Here is a picture of the program's window:

The root node for the window is a *VBox* containing four items, where the first three items are *HBoxes*. The first *HBox* contains two components, a *Label* displaying the text "`x =`" and a *TextField*. It is created with the commands

```
xInput = new TextField("0");  // Text input box initially
containing "0"
HBox xPane = new HBox( new Label(" x = "), xInput );
```

and it is later added as the first child of the *VBox* Note that the label is simply created with a constructor and added to the *HBox*, since there is no need to keep a reference to it.

The third *HBox* contains the four buttons. The buttons' default widths would be rather small. To make them fill the entire *HBox*, I add an hgrow constraint to each button and increase its maximum width. Here is how it's done:

```
/* Create the four buttons and an HBox to hold them. */

Button plus = new Button("+");
plus.setOnAction( e -> doOperation('+') );

Button minus = new Button("-");
minus.setOnAction( e -> doOperation('-') );

Button times = new Button("*");
times.setOnAction( e -> doOperation('*') );

Button divide = new Button("/");
divide.setOnAction( e -> doOperation('/') );

HBox buttonPane = new HBox( plus, minus, times, divide );

/* The four buttons need to be tweaked so that they will
fill
 * the entire buttonPane.  This can be done by giving each
button
 * a large maximum width and setting an hgrow constraint
 * for the button. */

HBox.setHgrow(plus, Priority.ALWAYS);
plus.setMaxWidth(Double.POSITIVE_INFINITY);
HBox.setHgrow(minus, Priority.ALWAYS);
minus.setMaxWidth(Double.POSITIVE_INFINITY);
HBox.setHgrow(times, Priority.ALWAYS);
times.setMaxWidth(Double.POSITIVE_INFINITY);
HBox.setHgrow(divide, Priority.ALWAYS);
divide.setMaxWidth(Double.POSITIVE_INFINITY);
```

The last position in the *VBox* is occupied by a *Label*. Since there is only one component, it is added directly to the *VBox*; there is no need to wrap it in an *HBox*. However, in order to get the text in the label to appear in the center instead of at the left edge of the window, I needed to increase the maximum

width of the label (so that the *VBox* will set its width to fill the entire available space). Furthermore, I had to set the alignment property of the label, to tell the label to place its text in the center of the label and not at the left:

```
answer.setMaxWidth(Double.POSITIVE_INFINITY);
answer.setAlignment(Pos.CENTER);
```

One point of interest in this example, aside from the layout, is the `doOperation()` method that is called when the user clicks one of the buttons. This method must retrieve the user's numbers from the text fields, perform the appropriate arithmetic operation on them (depending on which button was clicked), and set the text of the *Label* 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:

```
private void doOperation( char op ) {

    double x, y;  // The numbers from the input boxes.

    try {  // Get x from the first input box.
        String xStr = xInput.getText();
        x = Double.parseDouble(xStr);

    }
    catch (NumberFormatException e) {
            // The string xStr was not a legal number.
            // Show an error message, move the input focus
            // to xInput and select all the text in that
box.
        answer.setText("Illegal data for x.");
        xInput.requestFocus();
        xInput.selectAll();
        return; // stop processing when an error occurs!
    }

    try {  // Get a number from the second input box.
        String yStr = yInput.getText();
        y = Double.parseDouble(yStr);
    }
    catch (NumberFormatException e) {
        answer.setText("Illegal data for y.");
        yInput.requestFocus();
```

```
        yInput.selectAll();
        return;
    }

    /* Perform the operation based on the parameter, op.
*/

    if (op == '+')
        answer.setText( "x + y = " + (x+y) );
    else if (op == '-')
        answer.setText( "x - y = " + (x-y) );
    else if (op == '*')
        answer.setText( "x * y = " + (x*y) );
    else if (op == '/') {
        if (y == 0) {  // Can't divide by zero! Show an
error message.
            answer.setText("Can't divide by zero!");
            yInput.requestFocus();
            yInput.selectAll();
        }
        else {
            answer.setText( "x / y = " + (x/y) );
        }
    }

} // end doOperation()
```

The complete source code for this example can be found in SimpleCalc.java.

6.5.4 GridPane and TilePane

Next, we consider *GridPane*, a subclass of *Pane* that lays out its children in rows and columns. Rows and columns are numbered, starting from zero. Here is an illustration of a typical grid with 4 rows and 5 columns:

Note that the rows are not necessarily all the same height, and columns are not necessarily all the same width.

It is possible to leave gaps between rows or between columns, where the background of the *GridPane* will show. If `grid` is the *GridPane*, set the size of the gaps by calling

```
grid.setHGap( gapSize );  // gap between columns
gris.setVGap( gapSize );  // gap between rows
```

You can specify the row and column where a child is to be placed when you add the child to the *GridPane*:

```
grid.add( child, column, row );
```

(Note that the column number is specified first.) In fact, it is possible for a child to span several rows or several columns in the grid. The number of columns and number of rows that the child should occupy can be given as additional parameters to the `add()` method:

```
grid.add( child, column, row, columnCount, rowCount );
```

The number of rows and number of columns in the grid are determined by the positions of the children that have been added to it.

A *GridPane* will resize each child to fill the position or positions that it occupies in the grid (within minimum and maximum size limits). The preferred width of a column will be just large enough to accommodate the preferred widths of all the children in that column, and similarly for the preferred height. There are several ways to tweak this behavior, but here I will only mention that the actual column width and row height can be controlled.

A *GridPane* can have a set of **column constraints** and a set of **row constraints** that determine how the column width and row height are computed. The width of a column can be computed based on the widths of its children (the default), it can be set to a constant value, or it can be computed as a percentage of available space. Computation of row height is similar. Here is how to set constant row heights for a `gridpane` that has four rows:

```
gridpane.getRowConstraints().addAll(
     new RowConstraints(100), // row 0 has height 100
pixels
     new RowConstraints(150), // row 1 has height 150
pixels
     new RowConstraints(100), // row 2 has height 100
pixels
     new RowConstraints(200), // row 3 has height 200
pixels
);
```

In this case, the total height of the gridpane is the same, no matter how much space is available for it.

When percentages are used, the grid pane will expand to fill available space, and the row height or column width will be computed from the percentages. For example, to force a five-column gridpane to fill the available width and to force all columns to have the same size:

```
for (int i = 0; i < 5; i++) {
   ColumnConstraints constraints = new
ColumnConstraints();
   constraints.setPercentWidth(20); // (there is no
constructor that does this)
   gridpane.getColumnConstraints().add(constraints);
}
```

If the percentages add up to more than 100, then they will be decreased proportionately. In the example, setting all five percentages to be 50 would have the same effect as setting them all to be 20. Things can get more complicated than this, but these are probably the most common cases.

For an example of using a *GridPane* see the source code of SimpleColorChooser.java from Subsection 6.2.1. You should now be able to read and understand the entire program.

If you would like a grid in which all the rectangles are the same size, you can use a *TilePane*. A *TilePane* arranges equal-size "tiles" in rows and columns. Each tile holds one of the child nodes; there is no way to spread out a child over several tiles.

A *TilePane*, `tpane`, has a preferred number of rows and a preferred number of columns, which you should set by calling

```
tpane.setPrefColumns(cols);
```

The preferred number of columns will be used if the *TilePane* is shown at its preferred size, which is the usual case. However, that is not necessarily true if it is shown at a different size; in that case, the number of columns will be determined by the width of the available space. The number of rows will be determined by the number of child nodes that are added to the *TilePane*. The children will be laid out in order, filling the first row from left to right, then the second row, and so on. It is not required that the last row be completely filled. (All of this is true if the "orientation" of the *TilePane* is horizontal. It is possible to change the orientation to vertical. In that case, the number of preferred rows is specified instead of the number of columns. However, I have found little reason to do that.)

It's very common to have a *TilePane* in which the number of preferred columns is one. In that case, it is similar to a *VBox*. It is also common for the number of columns to be equal to the number of child nodes, and in that case, it is similar to an *HBox*.

At its preferred size, the width of each tile is equal to the largest preferred width of any child node, and similarly for the height. The *TilePane* will resize each child to completely fill its tile, except that it will not make a child larger than its maximum width and height.

There is a *TilePane* constructor with no parameters and one that takes a list of any number of children to be added to the pane. You can also add children later using one of

```
tpane.getChildren().add(child);
tpane.getChildren().addAll(child1, child2, ...);
```

You can specify the size of a horizontal gap that will be placed between columns and of a vertical gap that will be placed between rows. The background of the pane will show through the gaps. The gap amounts can be specified in the constructor (optionally followed by a list of children):

```
TilePane tpane = new TilePane( hgapAmount, vgapAmount );
```

or they can be specified later with `tpane.setHgap(h)` and `tpane.setVgap(v)`.

For an example that uses *TilePanes* for layout, see the solution to Exercise 6.7.

Section 6.6

Complete Programs

IN THIS CHAPTER, WE HAVE covered many of the basic aspects of GUI programming. There is still a lot more to learn, and we will return to the topic in Chapter 13. But you already know enough to write some interesting programs. In this section, we look at two complete programs that use what you have learned about GUI programming in this chapter, as well as what you learned about programming in general in earlier chapters. Along the way, we will also encounter a few new ideas.

6.6.1 A Little Card Game

The first program that we will consider is a GUI version of the command-line card game HighLow.java from Subsection 5.4.3. In the new version, HighLowGUI.java, 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.) In this GUI version of the program, 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 a game, you can click "New Game" to start another game. Here is what the program looks like in the middle of a game:

The complete source code for the program can be found in the file HighLowGUI.java. I encourage you to compile and run it. Note that the program also requires Card.java, Deck.java, and Hand.java, from Section 5.4, since they define classes that are used in the program. And it requires the file of card images, cards.png, that was used in RandomCards.java from Subsection 6.2.4.

The layout of the program should be easy to guess: `HighLowGUI` uses a *BorderPane* as the root of the scene graph. The center position is occupied by a *Canvas* on which the cards and a message are drawn. The bottom position contains an *HBox* that in turn contains three *Buttons*. In order to make the buttons fill the *HBox*, I set them all to have the same width, as discussed in Subsection 6.5.3. You can see all this in the `start()` method from the program:

```
public void start(Stage stage) {

    cardImages = new Image("cards.png");      // Load card
images.
    board = new Canvas(4*99 + 20, 123 + 80); // Space for
4 cards.

    Button higher = new Button("Higher");     // Create the
buttons, and
    higher.setOnAction( e -> doHigher() );    //    install
event handlers.
    Button lower = new Button("Lower");
    lower.setOnAction( e -> doLower() );
    Button newGame = new Button("New Game");
    newGame.setOnAction( e -> doNewGame() );

    HBox buttonBar = new HBox( higher, lower, newGame );

    higher.setPrefWidth(board.getWidth()/3.0);  // Make
each button fill
    lower.setPrefWidth(board.getWidth()/3.0);   //     1/3
of the width.
    newGame.setPrefWidth(board.getWidth()/3.0);

    BorderPane root = new BorderPane();  // Create the
scene graph root node.
    root.setCenter(board);
    root.setBottom(buttonBar);
```

```
    doNewGame();  // set up for the first game

    Scene scene = new Scene(root);  // Finish setting up
the scene and stage.
    stage.setScene(scene);
    stage.setTitle("High/Low Game");
    stage.setResizable(false);
    stage.show();

}  // end start()
```

Note that the event handlers call methods such as `doNewGame()` that are defined elsewhere in the program. The programming of those methods is a nice example of thinking in terms of a state machine. (See Subsection 6.3.6.) 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 program 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 program has three methods to respond to button clicks: `doHigher()`, `doLower()`, and `newGame()`. 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 be 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 redrawn so that the user can see that the state has changed. The complete `newGame()` method is as follows:

```
/**
 * Called by the start() method, and called by an event
handler if
 * the user clicks the "New Game" button.  Start a new
game.
 */
private 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!";
        drawBoard();
        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;
    drawBoard();
} // 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 redrawn to show the new state. Here is the `doHigher()` method:

```
/**
 * Called by an event handler 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.
 */
private 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!";
        drawBoard();
        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 +
".";
    }
    drawBoard();
} // end doHigher()
```

The `drawBoard()` method, which is responsible for drawing the content of the canvas, 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. The technique for drawing the individual cards was explained in Section 6.2. See the source code for the method definition.

6.6.2 Menus and Menubars

Our second example program, "MosaicDraw," is a kind of drawing program. The source code for the program is in the file MosaicDraw.java. The program also requires MosaicCanvas.java. Here is a half-size screenshot showing a sample drawing made with the program:

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.) The program has one feature that we have not encountered before: 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 swatch 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 *MosaicCanvas* class, a subclass of *Canvas* that is defined in MosaicCanvas.java. *MosaicCanvas* is a highly reusable class for representing mosaics of colored rectangles. It was also used behind the scenes in the sample program in Subsection 4.7.3. The *MosaicCanvas* class does not directly support drawing on the mosaic, but it does support setting the color of each individual square. The MosaicDraw program installs mouse handlers on the canvas. The handlers respond to `MousePressed` and `MouseDragged` events on the canvas by applying the currently selected tool to the canvas at the square that contains the mouse position. This is a nice example of applying event listeners to an object to do something that was not programmed into the object itself.

I urge you to study MosaicDraw.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 MosaicCanvas.java, it uses some techniques that you would not understand at this point, but I encourage you to at least read the comments in that file to learn about the API for *MosaicCanvas*.

MosaicDraw is the first example that we have seen that uses a menu bar. Fortunately, menus are very easy to use in JavaFX. The items in a menu are represented by objects belonging to class *MenuItem* or to one of its subclasses. (*MenuItem* and other menu-related classes are in package `javafx.scene.control`.) Menu items are used in almost exactly the same way as buttons. In particular, a *MenuItem* can be created using a constructor that specifies the text of the menu item, such as:

```
MenuItem fillCommand = new MenuItem("Fill");
```

Menu items, like buttons, can have a graphic as well as text, and there is a second constructor that allows you to specify both text and graphic. When the user selects a *MenuItem* from a menu, an *ActionEvent* is generated. Just as for a button, you can add an action event listener to the menu item using its `setOnAction(handler)` method. A menu item has a `setDisable(disabled)` method that can be used to enable and disable the item. And it has a `setText()` method for changing the text that is displayed in the item.

The main difference between a menu item and a button, of course, is that a menu item is meant to appear in a menu. (Actually, a menu item is a *Node* that can appear anywhere in a scene graph, but the usual place for it is in a menu.) A menu in JavaFX is represented by the class *Menu*. (In fact, *Menu* is actually a subclass of *MenuItem*, which means that you can add a menu as an item in another menu. The menu that you add becomes a submenu of the menu that you add it to.) A *Menu* has a name, which is specified in the constructor. It has an instance method `getItems()` that returns a list of menu items contained in the menu. To add items to the menu, you need to add them to that list:

```
Menu sampleMenu = new Menu("Sample");
sampleMenu.getItems().add( menuItem );  // Add one menu
item to the menu.
sampleMenu.getItems().addAll( item1, item2, item3 );  //
Add multiple items.
```

Once a menu has been created, it can be added to a menu bar. A menu bar is represented by the class *MenuBar*. A menu bar is just a container for menus. It does not have a name. The *MenuBar* constructor can be called with no parameters, or it can have a parameter list containing *Menus* to be added to the menu bar. The instance method `getMenus()` returns a list of menus, with methods `add()` and `addAll()` for adding menus to the menu bar. For example, the `MosaicDraw` program uses three menus, `controlMenu`, `colorMenu`, and `toolMenu`. We could create a menu bar and add the menus to it with the statements:

```
MenuBar menuBar = new MenuBar();
menuBar.getMenus().addAll(controlMenu, colorMenu,
toolMenu);
```

Or we could list the menus in the menu bar constructor:

```
MenuBar menuBar = new MenuBar(controlMenu, colorMenu,
toolMenu);
```

The final step in using menus is to add the menu bar to the program's scene graph. The menu bar could actually appear anywhere, but typically, it should be at the top of the window. A program that has a menu bar will usually use a *BorderPane* as the root of its scene graph, and it will add the menu bar as the top component in that root pane. The rest of the GUI for the program can be placed in the other four positions of the border pane.

So using menus generally follows the same 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). Place the menu bar at the top of a *BorderPane*, which is the root of the scene graph.

There are other kinds of menu items, defined by subclasses of *MenuItem*, that can be added to menus. A very simple example is *SeparatorMenuItem*, which appears in a menu as a line between other menu items. You can see an example in the "Control" menu of *MosaicDraw*. To add a separator to a *Menu*, `menu`, you just need to say

```
menu.getItems().add( new SeparatorMenuItem() );
```

Much more interesting are the subclasses *CheckMenuItem* and *RadioMenuItem*.

A *CheckMenuItem* represents a menu item that can be in one of two states, selected or not selected. The state is changed when the user selects the item from the menu that contains it. A *CheckMenuItem* has the same functionality and is used in the same way as a *CheckBox* (see Subsection 6.4.3). Three *CheckMenuItems* are used in the "Control" menu of the MosaicDraw program. One is 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 *CheckMenuItem* shows and hides "grouting"

in the mosaic (grouting consists of gray lines drawn around each of the little squares in the mosaic). The *CheckMenuItem* that corresponds to the "Use Randomness" option in the "Control" menu could be set up with the statements:

```
useRandomness = new CheckMenuItem("Use Randomness");
useRandomness.setSelected(true);  // Randomness is
initially turned on.
controlMenu.getMenus().add(useRandomness);   // Add menu
item to the menu.
```

No *ActionEvent* handler is added to `useRandomness`; the program simply checks its state by calling `useRandomness.isSelected()` whenever it is coloring a square, to decide whether to add some random variation to the color. On the other hand, when the user selects the "Use Grouting" check box from the menu, the canvas must immediately be redrawn to reflect the new state. A handler is added to the *CheckMenuItem* to take care of that by calling an appropriate method:

```
useGrouting.setOnAction( e ->
doUseGrouting(useGrouting.isSelected()) );
```

The "Color" and "Tools" menus contain items of type *RadioMenuItem*, which are used in the same way as the *RadioButtons* that were discussed in Subsection 6.4.3: A *RadioMenuItem*, like a check box, can be either selected or unselected, but when several *RadioMenuItems* are added to a *ToggleGroup*, then at most one of the group members can be selected. In the program, the user selects the tool that they want to use from the "Tools" menu. Only one tool can be selected at a time, so it makes sense to use *RadioMenuItems* to represent the available tools, and to put all of those items into the same *ToggleGroup*. The currently selected option in the "Tools" menu will be marked as selected; when the user chooses a new tool, the mark is moved. This gives the user some visible feedback about which tool is currently selected for use. Furthermore, the *ToggleGroup* has an observable property representing the currently selected option (see Subsection 6.3.7). The program adds a listener to that property with an event handler that will be called whenever the user selects a new tool. Here is the code that creates the "Tools" menu:

```
Menu toolMenu = new Menu("Tools");
ToggleGroup toolGroup = new ToggleGroup();
toolGroup.selectedToggleProperty().addListener(
                 e ->
doToolChoice(toolGroup.getSelectedToggle()) );
addRadioMenuItem(toolMenu,"Draw",toolGroup, true);
addRadioMenuItem(toolMenu,"Erase",toolGroup, false);
addRadioMenuItem(toolMenu,"Draw 3x3",toolGroup, false);
addRadioMenuItem(toolMenu,"Erase 3x3",toolGroup, false);
```

The `addRadioMenuItem` method that is used in this code is a utility method that is defined elsewhere in the program:

```
/**
 * Utility method to create a radio menu item, add it
 * to a ToggleGroup, and add it to a menu.
 */
private void addRadioMenuItem(Menu menu, String command,
                                    ToggleGroup group,
boolean selected) {
    RadioMenuItem menuItem = new RadioMenuItem(command);
    menuItem.setToggleGroup(group);
    menu.getItems().add(menuItem);
    if (selected) {
        menuItem.setSelected(true);
    }
}
```

The complete code for creating the menu bar in `MosaicDraw` can be found in a method `createMenuBar()`. Again, I encourage you to study the source code.

6.6.3 Scene and Stage

Before ending this brief introduction to GUI programming, we look at two fundamental classes in a little more detail: *Scene*, from package `javafx.scene`, and *Stage*, from package `javafx.stage`.

A *Scene* represents the content area of a window (that is, not including the window's border and title bar), and it serves as a holder for the root of the scene graph. The *Scene* class has several constructors, but they all require

the root of the scene graph as one of the parameters, and the root cannot be `null`. Perhaps the most common constructor is the one that has only the root as parameter: `new Scene(root)`.

A scene has a width and a height, which can be specified as parameters to the constructor: `new Scene(root,width,height)`. In the typical case where the `root` is a *Pane*, the size of the pane will be set to match the size of the scene, and the pane will lay out its contents based on that size. If the size of the scene is not specified in the constructor, then the size of the scene will be set to the preferred size of the pane. It is not possible for a program to set the width or height of a *Scene* after it has been created, but if the size of the stage that contains a scene is changed, then the size of the scene is automatically changed to match the new size of the stage's content area, and the root node of the scene (if it is a *Pane*) will be resized as well.

A *Scene* can have a background fill *color* (actually a *Paint*), which can be specified in the constructor. Generally, the scene's background is not seen, since it is covered by the background of the root node. The default style sets the background of the root to be light gray. However, you can set the background color of the root to be transparent if you want to see the scene background instead.

A *Stage*, from package `javafx.stage`, represents a window on the computer's screen. Any JavaFX *Application* has at least one stage, called the primary stage, which is created by the system and passed as a parameter to the application's `start()` method. Although we have not seen any examples so far in this textbook, many programs use more than one window. It is possible for a program to create new *Stage* objects; we will see how to do that in Chapter 13.

A stage contains a scene, which fills its content area. The scene is installed in the stage by calling the instance method `stage.setScene(scene)`. It is possible to show a stage that does not contain a scene, but its content area will just be a blank rectangle.

In addition to a content area, a stage has a title bar above the content. The title bar contains a title for the window and some "decorations"—little

controls that the user can click to do things like close and maximize the window. The title bar is provided by the operating system, not by Java, and its style is set by the operating system. The instance method `stage.setTitle(string)` sets the text that is shown in the title bar. The title can be changed at any time.

By default a stage is resizable. That is, the size of the window can be changed by the user, by dragging its borders or corners. To prevent the user from changing the window size, you can call `stage.setResizable(false)`. However, a program can change the size of a stage with the instance methods `stage.setWidth(w)` and `stage.setHeight(h)`, and this can be done even if the stage has been made non-resizable. Usually, the initial size of a stage is determined by the size of the scene that it contains, but it is also possible to set the initial size before showing the window using `setWidth()` and `setHeight()`.

By default, when a stage is resizable, the user can make the window arbitrarily small and arbitrarily large. It is possible to put limits on the resizability of a window with the instance methods `stage.setMinWidth(w)`, `stage.setMaxWidth(w)`, `stage.setMinHeight(h)`, and `stage.setMaxHeight(h)`. The size limits apply only to what the user can do by dragging the borders or corners of the window.

It is also possible to change the position of a stage on the screen, using the instance methods `stage.setX(x)` and `stage.setY(y)`. The x and y coordinates specify the position of the top left corner of the window, in the coordinate system of the screen. Typically, you would do this before showing the stage.

Finally, for now, remember that a stage is not visible on the screen until you show it by calling the instance method `stage.show()`. Showing the primary stage is typically the last thing that you do in a application's `start()` method.

6.6.4 Creating Jar Files

Java classes and resource files are often distributed in jar ("java archive") files. For a program that consists of multiple files, it can make sense to pack them into a single jar file. As the last topic for this chapter, we look at how to do that. The program can be run directly from the jar file, without unpacking it. However, for JavaFX programs, the user will still need access to the JavaFX SDK. A jar file can be "executable," meaning that it specifies the class that contains the `main()` routine that will be run when the jar file is executed. If you have an executable jar file that does not require JavaFX or other external resources, you can run it on the command line using a command of the form:

```
java  -jar  JarFileName.jar
```

and you might even be able to run the jar file by double-clicking its icon in a file browser window. If an executable jar file requires JavaFX, you will need to add the same options to the `java` command that were discussed in Subsection 2.6.7. For example,

```
java -p /opt/jfx17/lib --add-modules=ALL-MODULE-PATH
JarFileName.jar
```

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 Development 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 and resource files that you want to include). 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 full path 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. 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.

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  -c  -f  JarFileName.jar  *.class
```

where `JarFileName` can be any name that you want to use for the jar file. The `-c` option is used to create a jar file. The `-f` is followed by the name of the jar file that is to be created. 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. You can also list the class file names separately. If the program uses resource files, such as images, they should also be listed in the command. (Things get more complicated if your classes and resources are not in the default package. In that case, the files must be in subdirectories of the directory in which you issue the `jar` command, and you have to include the path to the file in the name. For example: `textio/TextIO.class` on MacOS and Linux, or `textio\TextIO.class` on Windows.)

Making an executable jar file on the command line is only a little more complicated. There has to be some way of specifying which class contains the `main()` routine. This can be done by adding the `-e` option to the command, with a value giving the full name of the class that is to be executed when the jar file is run. For example, if the name of the class is *MyMainClass*, then the jar file could be created with:

```
jar  -c  -f  JarFileName.jar  -e  MyMainClass  *.class
```

For a program defined in two packages, `grapher.ui` and `grapher.util`, with a main class defined in the file `Main.java` in package `grapher.ui`, the command would become

```
jar -c -f Grapher.jar -e grapher.ui.Main
grapher/ui/*.class grapher/util/*.class
```

except that on Windows, the slashes would be replaced by backslashes.

(The options `-c`, `-f`, and `-e` are abbreviations for the longer forms `--create`, `--file`, and `--main-class`, and you can use the longer forms, if you prefer clarity to brevity.)

6.6.5 jpackage

You can collect the class files and resource files for a program into a jar file, and you can give that jar file to someone who would like to use your program. However, that person will still need to have Java installed on their computer—something that is really not very likely these days, unless that person is a Java developer. A possible solution is to bundle a Java virtual machine along with your program. The Java Development Kit includes the `jpackage` command to make that possible.

The `jpackage` command can create an installer that will install your program along with as much Java support as is needed to run it. It has some significant limitations. It can only make an installer for the type of computer and operating system that you are using; for example, it is not possible to use `jpackage` on Linux to make an installer for Windows. And the files that it makes are very large, since they have to include large parts of a Java execution environment. So `jpackage` is really meant for packaging large, serious applications. But if you want to try it, here is a basic example, using only a few of the options that are available for `jpackage`. For the example, I made an installer for the network poker game from Subsection 12.5.4. This might also help you understand how to work with packages in general.

To apply `jpackage` to a program that uses JavaFX, you have to make Java packages available to the `jpackage` command. For this purpose, you can't

use the JavaFX SDK. Instead, you need the JavaFX "jmods". See Subsection 2.6.7 for a discussion of the JavaFX SDK. For `jpackage`, you will need to download the jmods for your operating system from https://gluonhq.com/products/javafx/. For my computer, I extracted the downloaded file into `/opt/javafx-jmods-17.0.2`. (The jmods are required because they include specific operating system support that is not in the SDK jar files.)

To use `jpackage`, you need a jar file that contains the classes and resource files for your program. The poker game uses classes from the packages `netgame.common` and `netgame.fivecarddraw`, plus a resource image file `cards.png` in `netgame.fivecarddraw`. To make the jar file, I first compiled the Java files for the program. Since the poker game uses JavaFX, I included the necessary options in the `javac` command. I used the following command in the directory that contained the netgame directory, *typed all on one line*:

```
javac  --module-path  /opt/jfx17/lib  --add-modules=ALL-
MODULE-PATH
                    netgame/common/*.java
netgame/fivecarddraw/*.java
```

I then created a jar file, `Poker.jar`, with this command, *typed all on one line*:

```
jar -c -f Poker.jar  netgame/common/*.class
netgame/fivecarddraw/*.class
                netgame/fivecarddraw/cards.png
```

It is important to include the image resource file along with the class files. (Note that on Windows, the slashes, "/", would be replaced by backslashes, "\".)

I moved `Poker.jar` to a new directory. Working in that new directory, I used the following very long `jpackage` command, again *typed all on one line*:

```
jpackage  --input  .  --main-jar  Poker.jar
            --main-class  netgame.fivecarddraw.Main  --
name  NetPoker
            --module-path /opt/javafx-jmods-17.0.2
```

```
            --add-modules
javafx.base,javafx.controls,javafx.graphics
```

The value for the `--input` option is a period, which represents the current working directory; it could be replaced by a path to the directory that contains `Poker.jar`. The value for the `--main-class` option is the full name of the class that contains the main program; this option is not needed if the jar file is executable. The value of the `--name` option is used to name the installer and to name the executable file that it will install. The `--module-path` refers to the directory that contains the JavaFX jmods. And the added modules are just those JavaFX modules that are needed for this program.

When used on my computer, running Linux Mint, this produced a file named `netpoker_1.0-1_amd64.deb` that I could then install in the usual way. It installed the poker executable as `/opt/netpoker/bin/NetPoker`.

On MacOS 10.15, using the JDK from adoptium.net (see Subsection 2.6.1), I found that `jpackage` was installed as part of the JDK, but it was not made available on the command-line. I was able to define it myself as an alias:

```
alias jpackage=
     "/Library/Java/JavaVirtualMachines/temurin-
17.jdk/Contents/Home/bin/jpackage"
```

Again, type this all on one line. With that done, I used the same commands as given above, with appropriate directory names for the JavaFX SDK and jmods. The result was a .dmg file containing a program, `netpoker.app`, that could be run by double-clicking. (The jpackage command might be properly set up in newer versions of MacOS.)

The `jpackage` command should also work on Windows, but it requires something called the "WiX toolset" in addition to the JDK. I have not tried it.

Programming Exercises for Chapter 6

Exercise 6.1:

In Subsection 6.3.2, the following code was given as an example. It installs a `MousePressed` event handler on a canvas. The handler lets the user draw a red rectangle at the point where the user clicks the mouse, or, by holding the shift key down, a blue oval:

```
canvas.setOnMousePressed( evt -> {
    GraphicsContext g = canvas.getGraphicsContext2D();
    if ( evt.isShiftDown() ) {
        g.setFill( Color.BLUE );
        g.fillOval( evt.getX() - 30, evt.getY() - 15, 60,
30 )
    }
    else {
        g.setFill( Color.RED );
        g.fillRect( evt.getX() - 30, evt.getY() - 15, 60,
30 );
    }
} );
```

Write a complete program that does the same, but in addition, the program will continue to draw figures if the user drags the mouse. That is, the mouse will leave a trail of figures as the user drags. However, if the user right-clicks the canvas, then the canvas should simply be cleared and no figures should be drawn even if the user drags the mouse after right-clicking. See the discussion of dragging in Subsection 6.3.3. Here is a picture of my solution:

Note that a black border has been added around each shape to make them more distinct.

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 where the previous shape was drawn.

See the Solution

Exercise 6.2:

Write a program 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 out of the window if she wants, and it will disappear. To allow the user to get the squares back into the window, add a `KeyPressed` event handler that will restore the squares to their original positions when the user presses the Escape key. The key code for the Escape key is `KeyCode.ESCAPE`.

See the Solution

Exercise 6.3:

Write a program that shows a pair of dice. The dice are drawn on a *Canvas*. You can assume that the size of the canvas is 100 by 100 pixels. When the user clicks on the canvas, 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, such as "`void drawDie(GraphicsContext g, int val, int x, int y)`", to draw a die at the specified `(x,y)` coordinates. The second parameter, `val`, specifies the number of dots that are showing on the die. Here is a picture of a canvas displaying two the dice:

See the Solution

Exercise 6.4:

In Exercise 6.3, you wrote a graphical pair-of-dice program where the dice are rolled when the user clicks on the canvas. Now make a pair-of-dice program where the user rolls the dice by clicking a button. The button should appear under the canvas 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.

See the Solution

Exercise 6.5:

In Exercise 3.8, you drew a checkerboard. For this exercise, write a program where the user can select a square by clicking on it. Highlight the selected square by drawing a colored border around it. When the program starts, 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 checkerboard is exactly 400 by 400 pixels, so that each square on the checkerboard is 50 by 50 pixels. Here is my checkerboard, with the square in row 3, column 3 selected, shown at reduced size:

See the Solution

Exercise 6.6:

For this exercise, you should modify the SubKiller game from Subsection 6.3.6. You can start with the existing source code, from the file SubKiller.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 canvas 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.

See the Solution

Exercise 6.7:

Exercise 5.2 involved a class, StatCalc.java, that could compute some statistics of a set of numbers. Write a GUI program that uses the *StatCalc* class to compute and display statistics of numbers entered by the user. The program will have an instance variable of type *StatCalc* that does the computations. The program should include a *TextField* 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 *TextField* and then either clicking an "Enter" button or pressing the Return (or Enter) key. There should be a "Clear" button that clears out all the data. This means creating a new *StatCalc* object and changing the text that is displayed on the labels. (See the discussion of "default buttons" at the end of Subsection 6.4.2 for one way of implementing a response the Return key.) Here is a picture of my solution to this problem:

Getting the interface to look just like I wanted it was the hardest part. In the end, I used *TilePanes* (Subsection 6.5.4) for the layout.

See the Solution

Exercise 6.8:

Write a program that has a *TextArea* where the user can enter some text. The program should have a button such that when the user clicks on the button, the program will 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*. Recall that if `textInput` is a *TextArea*, then you can get the contents of the *TextArea* 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. Here is a picture of my solution:

See the Solution

Exercise 6.9:

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. Subsection 6.2.4 has a list of shape-drawing methods in a *GraphicsContext*. Among them are methods for stroking and for filling polygons: `g.strokePolygon(xcoords,ycoords,n)` and `g.fillPolygon(xcoords,ycoords,n)`, For these commands, the coordinates of the vertices of the polygon are stored in arrays `xcoords` and `ycoords` of type `double[]`, and the number of vertices of the polygon is given by the third parameter, `n`. Note that it is OK for the sides of a polygon to cross each other, but the interior of a polygon with self-intersections might not be exactly what you expect.

Write a program that lets the user draw polygons. As the user clicks a sequence of points in a *Canvas*, count the points and store their x- and y-coordinates in two arrays. These points will be the vertices of the polygon. As the user is creating the polygon, you should just connect all the points with line segments. When the user clicks near the starting point, draw the complete polygon. Draw it with a red interior and a black border. Once the user has completed a polygon, the next click should clear the data and start a new polygon from scratch.

Here is a picture of my solution after the user has drawn a fairly complex polygon:

See the Solution

Exercise 6.10:

Write a GUI Blackjack program that lets the user play a game of Blackjack, with the computer as the dealer. The program should draw the user's cards and the dealer's cards, just as was done for the graphical HighLow card game in Subsection 6.6.1. 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 program and the Blackjack program 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 make the canvas just wide enough to show five cards, and tall enough to show both 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. It will also need the images file cards.png, which contains pictures of the cards.

The next exercise has a picture of a Blackjack game that you can use a guide, except that the version for this exercise does not allow betting. (Some aesthetic changes to the GUI were made in that Blackjack program, compared to the HighLow program.)

See the Solution

Exercise 6.11:

In the Blackjack game from Exercise 6.10, 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 method that can be called every time it is necessary to set the value of the `gameInProgress` variable. That method can take full responsibility for enabling and disabling the buttons (as long as it is used consistently). Recall that if `bttn` is a variable of type `Button`, then `bttn.setDisable(true)` disables the button and `bttn.setDisable(false)` 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 program starts, give the user $100. Add a *TextField* to the strip of controls along the bottom of the panel. The user enters the bet in this *TextField*. 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 *TextField* might not be a legal number, the bet that the user places might be more money than the user has, or the bet 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 *TextField* uneditable while the game is in progress. If `betInput` is the *TextField*, you can make it editable and uneditable by the user with the commands `betInput.setEditable(true)` and `betInput.setEditable(false)`.

In the `drawBoard()` 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 program 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 `start()` method, 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:

See the Solution

Quiz on Chapter 6

Question 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.

Question 2:

A central concept in JavaFX is the *scene graph*? What is a scene graph?

Question 3:

JavaFX has standard classes called *Canvas* and *GraphicsContext*. Explain the purpose of these classes.

Question 4:

Suppose that `canvas` is a variable of type *Canvas*. Write a code segment that will draw a green square centered inside a red circle on the canvas, as illustrated.

Question 5:

Draw the picture that will be produced by the following `for` loop, where `g` is a variable of type *GraphicsContext*:

```
for (int i=10; i <= 210; i = i + 50) {
   for (int j = 10; j <= 210; j = j + 50) {
      g.strokeLine(i,10,j,60);
```

```
    }
}
```

Question 6:

JavaFX has a standard class called *MouseEvent*. What is the purpose of this class? What does an object of type *MouseEvent* do?

Question 7:

Explain the difference between a `KeyPressed` event and a `KeyTyped` event.

Question 8:

Explain how *AnimationTimers* are used to do animation.

Question 9:

What is a *CheckBox* and how is it used? How does it differ from a *RadioButton*?

Question 10:

What is meant by *layout* of GUI components? Explain briefly how layout is done in JavaFX.

Question 11:

How does a *BorderPane* arrange the components that it contains?

Question 12:

How is the *preferred size* of a component set, and how is it used?

See the Answers

Chapter 7

Arrays, ArrayLists, and Records

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, combined usually with some methods, 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. This chapter looks at more basic data structures. There is one type of data structure that is so important and so basic that it is built into every programming language: the array.

You have already encountered arrays in Section 3.8 and Subsection 5.1.4. We continue the study of arrays in this chapter, including some new details of their use and some additional array-processing techniques. In particular, we will look at the important topic of algorithms for searching and sorting an array in Section 7.5.

An array has a fixed size that can't be changed after the array is created. But in many cases, it is useful to have a data structure that can grow and shrink as necessary. In Section 7.3, we will look at a standard class, *ArrayList*, that represents such a data structure.

An array is a numbered sequence of items, all of the same type. A **record** is another kind of standard data structure. Like an array, a record consists of a sequence of items, but in a record, the items are referred to by name instead of by number, and the items can be of different types. Any object in Java can be thought of as a record in this sense, but Java 17 introduced records explicitly into the language as a special kind of class with certain restrictions. We will look at record classes in Section 7.4.

Contents of Chapter 7:

- Section 1: Array Details
- Section 2: Array Processing
- Section 3: ArrayList
- Section 4: Records
- Section 5: Searching and Sorting
- Section 6: Two-dimensional Arrays
- Programming Exercises
- Quiz on This Chapter

Section 7.1

Array Details

ARRAY BASICS HAVE BEEN DISCUSSED in previous chapters, but there are still details of Java syntax to be filled in, and there is a lot more to say about using arrays. This section looks at some of the syntactic details, with more information about array processing to come in the rest of the chapter.

To briefly review some of the basics.... An array is a numbered sequence of *elements*, and each element acts like a separate variable. All of the elements are of the same type, which is called the *base type* of the array. The array as a whole also has a type. If the base type is *btype*, then the array is of type `btype[]`. Each element in the array has an *index*, which is just its numerical position in the sequence of elements with numbering starting from zero. If the array is `A`, then element number `i` of the array is `A[i]`. The number of elements in an array is called its *length*. The length of an array `A` is `A.length`. The length of an array can't be changed after the array is created. The elements of the array `A` are `A[0]`, `A[1]`, ..., `A[A.length-1]`. An attempt to refer to an array element with an index outside the range from zero to `A.length-1` causes an *ArrayIndexOutOfBoundsException*.

Arrays in Java are objects, so an array variable can only **refer** to an array; it does **not** contain the array. The value of an array variable can also be `null`. In that case, it does not refer to any array, and an attempt to refer to an array element such as `A[i]` will cause a *NullPointerException*. Arrays are created using a special form of the `new` operator. For example,

```
int[] A = new int[10];
```

creates a new array with base type int and length 10, and it sets the variable `A` to refer to the newly created array.

7.1.1 For-each Loops

Arrays are often processed using `for` loops. A `for` loop makes it easy to process each element in an array from beginning to end. For example, if `namelist` is an array of *Strings*, then all the values in the list can be printed using

```
for (int i = 0; i < namelist.length; i++) {
    System.out.println( namelist[i] );
}
```

This type of processing is so common that there is an alternative form of the `for` loop that makes it easier to write. The alternative is called a **for-each loop**. It is probably easiest to start with an example. Here is a for-each loop for printing all the values in an array of *Strings*:

```
for ( String name : namelist ) {
    System.out.println( name );
}
```

The meaning of "`for (String name : namelist)`" is "for each String, name, in the array, namelist, do the following". The effect is that the variable `name` takes on each of the values in `namelist` in turn, and the body of the loop is executed for each of those values. Note that there is no array index in the loop. The loop control variable, `name`, represents one of the values in the array, not the index of one of the values.

The for-each loop is meant specifically for processing all the values in a data structure, and we will see in Section 7.3 and Chapter 10 that it applies to other data structures besides arrays. The for-each loop makes it possible to process the values in a data structure without even knowing the details of how the data is structured. In the case of arrays, it lets you avoid the complications of using array indices.

A for-each loop will perform the same operation for each value that is stored in an array. If `itemArray` is an array of type `BaseType[]`, then a for-each loop for `itemArray` has the form:

```
for ( BaseType item : itemArray ) {
   .
   .  // process the item
```

```
    .
}
```

As usual, the braces are optional if there is only one statement inside the loop. In this loop, `item` is the loop control variable. It is 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; it cannot be a variable that already exists outside 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 < itemArray.length; index++ ) {
   BaseType item;
   item = itemArray[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;
}
```

I also note that the use of **var** for declaring local variables, which was introduced in Subsection 4.8.2, applies to the loop control variable in a for-each loop. So, instead of "`for (BaseType item : itemArray)`", we could write "`for (var item : itemArray)`". The type of the variable is deduced from the base type of the array. This syntax becomes more useful when dealing with more complicated types.

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, or to process the elements in reverse order. However, it does make the code a little simpler when you do want to process all the values, in order. since it eliminates any need to use array indices.

But 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 and that holds the value). 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. The loop is equivalent to

```
int[] intList = new int[10];
for ( int i = 0; i < intList.length; i++ ) {
   int item = intList[i];
   item = 17;
}
```

which certainly does not change the value of any element in the array.

7.1.2 Variable Arity Methods

Before Java 5, every method in Java had a fixed arity. (The **arity** of a method 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 and must be the same as the number of formal parameters in the

method's definition. Java 5 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 2.4.1, 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`, is what makes this a variable arity method. It 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 ) {
        // Inside this method, numbers is of type
double[].
   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; // (Will be Double.NaN if
numbers.length is zero.)
}
```

By the way, it is possible to pass a single array to a variable arity method, instead of a list of individual values. For example, suppose that `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.

The formal parameter list in the definition of a variable-arity method can include more than one parameter, but the `...` can only be applied to the very last formal parameter.

As an 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 double. 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. Remember that inside the definition of the method, the parameter `points` becomes an array of *Points*:

```
public static void drawPolygon(GraphicsContext 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.strokeLine( points[i].x, points[i].y,
points[i+1].x, points[i+1].y );
       }
       // Now, draw a line back to the starting point.
       g.strokeLine( points[points.length-1].x,
points[points.length-1].y,
```

```
                                points[0].x, points[0].y );
        }
}
```

When this method is called, the subroutine call statement must have one actual parameter of type *GraphicsContext*, which can be followed by any number of actual parameters of type *Point*.

For a final example, let's look at a method that strings together all of the values in a list of strings into a single, long string. This example uses a for-each loop to process the array:

```
public static String concat( String... values ) {
   StringBuilder buffer;  // Use StringBuilder for more
efficient concatenation.
   buffer = new StringBuilder();  // Start with an empty
StringBuilder.
   for ( String str : values ) { // A "for each" loop for
processing the values.
       buffer.append(str); // Add string to the buffer.
   }
   return buffer.toString(); // return the contents of the
buffer
}
```

Given this method definition, the method call `concat("Hello", "World")` would return the string "HelloWorld", and `concat()` would return an empty string. Since a variable arity method can also accept an array as actual parameter, we could also call `concat(lines)` where `lines` is of type `String[]`. This would concatenate all the elements of the array into a single string.

7.1.3 Array Literals

We have seen that it is possible to initialize an array variable with a list of values at the time it is declared. For example,

```
int[] squares = { 1, 4, 9, 16, 25, 36, 49 };
```

This initializes `squares` to refer to a newly created array that contains the seven values in the list.

A list initializer of this form can be used **only** in a declaration statement, where it gives 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 already existed. However, there is another, similar notation for creating a new array that can be used in other places. The notation uses another form of the `new` operator to both create a new array object and fill it with values. (The rather odd syntax is similar to the syntax for anonymous inner classes, which were discussed in Subsection 5.8.3.) As an example, to assign a new value to an array variable, `cubes`, of type `int[]`, you could use:

```
cubes = new int[] { 1, 8, 27, 64, 125, 216, 343 };
```

This is an assignment statement rather than a declaration, so the array initializer syntax, without "`new int[]`," would not be legal here. 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. In this sense, it is an "array literal," since it is something that you can type in a program to represent a value. This means that it can be used in any context where an object of type **base-type** `[]` is legal. For example, you could pass the newly created array as an actual parameter to a subroutine. Consider the following utility method for creating a menu from an array of strings. (Menus were discussed in Subsection 6.6.2.)

```
/**
 * Creates a Menu.  The names for the MenuItems in the
menu are
 *     given as an array of Strings.
 * @param menuName  the name for the Menu that is to be
created.
 * @param itemNames  an array holding the text that
appears in each
 *     MenuItem.  If a null value appears in the array,
the corresponding
 *     item in the menu will be a separator rather than a
MenuItem.
```

```
 * @return  the menu that has been created and filled with
items.
 */
public static Menu createMenu( String menuName, String[]
itemNames ) {
    Menu menu = new Menu(menuName);
    for ( String itemName : itemNames ) {
        if ( itemName == null ) {
            menu.getItems().add( new SeparatorMenuItem()
);
        }
        else {
            MenuItem item = new MenuItem(itemName);
            menu.getItems().add(item);
        }
    }
    return menu;
}
```

The second parameter in a call to `createMenu` is an array of strings. The array that is passed as an actual parameter could be created in place, using the `new` operator. For example, we can use the following statement to create an entire File menu:

```
Menu fileMenu = createMenu( "File",
               new String[] { "New", "Open", "Close", null,
"Quit" } );
```

This should convince you that being able to create and use an array "in place" in this way can be very convenient, in the same way that anonymous inner classes are convenient. (However, this example could have been done even more conveniently if `createMenu()` had been written as a variable arity method!)

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 sometimes 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.

Section 7.2

Array Processing

MOST EXAMPLES OF ARRAY PROCESSING that we have looked at have actually been fairly straightforward: processing the elements of the array in order from beginning to end, or random access to an arbitrary element of the array. In this section and later in the chapter, you'll see some of the more interesting things that you can do with arrays.

7.2.1 Some Processing Examples

To begin, here's an example to remind you to be careful about avoiding array indices outside the legal range. Suppose that `lines` is an array of type `String[]`, and we want to know whether `lines` contains any duplicate elements in consecutive locations. That is, we want to know whether `lines[i].equals(lines[i+1])` for any index `i`. Here is a failed attempt to check that condition:

```
boolean dupp = false;  // Assume there are no duplicates
for ( int i = 0; i < list.length; i++ ) {
    if ( lines[i].equals(lines[i+1]) ) {  // THERE IS AN
ERROR HERE!
        dupp = true;   // we have found a duplicate!
        break;
    }
}
```

This `for` loop looks like many others that we have written, so what's the problem? The error occurs when `i` takes on its final value in the loop, when `i` is equal to `lines.length-1`. In that case, `i+1` is equal to `lines.length`. But the last element in the array has index `lines.length-1`, so `lines.length` is not a legal index. This means that the reference to `lines[i+1]` causes an ArrayIndexOutOfBoundsException. This is easy to fix; we just need to stop the loop before `i+1` goes out of range:

```
boolean dupp = false;  // Assume there are no duplicates
for ( int i = 0; i < list.length - 1 ; i++ ) {
    if ( lines[i].equals(lines[i+1]) ) {
        dupp = true;   // we have found a duplicate!
        break;
    }
}
```

This type of error can be even more insidious when working with partially full arrays (see Subsection 3.8.4), where usually only part of the array is in use, and a counter is used to keep track of how many spaces in the array are used. With a partially full array, the problem is not looking beyond the end of the array, but looking beyond *the part of the array that is in use*. When your program tries to look beyond the end of an array, at least the program will crash to let you know that there is a problem. With a partially full array, the problem can go undetected.

For the next example, let's continue with partially full arrays. We have seen how to add an item to a partially full array, but suppose that we also want to be able to remove items? 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 follow the partially full array pattern, and you will 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: Besides

being the number of items in the array, `playerCt` is also the index of the next open spot in the array. 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.
```

But deleting a player from the game is a little harder, since you don't want to leave a "hole" in the array where the deleted player used to be. Suppose you want to delete the player at index `k` in `playerList`. The number of players goes down by one, so one fewer space is used in the array. If you are not worried about keeping the players in any particular order, then one way to delete player number `k` 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` has been replaced and is no longer in the array, so we have deleted that player from the list. 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. You can set `playerList[playerCt] = null`, which could free up the deleted *Player* object for garbage collection, but that is not necessary to correctly delete the player from the list of (active) players. (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 following must move up 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--;
```

Here is an illustration of the two ways of deleting an item from a partially full array. Here, player "C" is being deleted:

This leaves open the question of what happens when a partially full array becomes full, but you still want to add more items to it? We can't change the size of the array—but we can make a new, bigger array and copy the data from the old array into the new array. But what does it mean to copy an array in the first place?

Suppose that `A` and `B` are array variables, with the same base type, and that `A` already refers to an array. Suppose that we want `B` to refer to a copy of `A`. The first thing to note is that the assignment statement

```
B = A;
```

does **not** make a copy of `A`. Arrays are objects, and an array variable can only hold a pointer to an array. The assignment statement copies the pointer from `A` into `B`, and the result is that `A` and `B` now point to the same array. For example, `A[0]` and `B[0]` are just different names for exactly the same array

element. To make `B` refer to a copy of `A`, we need to make an entirely new array and copy all the items from `A` into `B`. Let's say that `A` and `B` are of type `double[]`. Then to make a copy of `A`, we can say

```
double[] B;
B = new double[A.length];  // Make a new array with the
same length as A.
for ( int i = 0; i < A.length; i++ ) {
    B[i] = A[i];
}
```

To solve the problem of adding to a partially full array that has become full, we just need to make a new array that is bigger than the existing array. The usual choice is to make a new array twice as big as the old. We need to meet one more requirement: At the end, the variable that referred to the old array must now point to the new array. That variable is what gives us access to the data, and in the end, the data is in the new array. Fortunately, a simple assignment statement will make the variable point to the correct array. Let's suppose that we are using `playerList` and `playerCt` to store the players in a game, as in the example above, and we want to add `newPlayer` to the game. Here is how we can do that even if the `playerList` array is full:

```
if ( playerCt == playerList.length ) {
        // The number of players is already equal to the
size of the array.
        // The array is full.  Make a new array that has
more space.
    Player[] temp;   // A variable to point to the new
array.
    temp = new Player[ 2*playerList.length ];  // Twice as
big as the old array.
    for ( int i = 0; i < playerList.length; i++ ) {
        temp[i] = playerList[i];  // Copy item from old
array into new array.
    }
    playerList = temp;  // playerList now points to the
new, bigger array.
}
// At this point, we know that there is room in the array
for newPlayer.
playerList[playerCt] = newPlayer;
playerCt++;
```

After the new array has been created, there is no longer any variable that points to the old array, so it will be garbage collected.

7.2.2 Some Standard Array Methods

Copying an array seems like such a common method that you might expect Java to have a built-in method already defined to do it. In fact, Java comes with several standard array-processing methods. The methods are defined as `static` methods in a class named *Arrays*, which is in the package `java.util`. For example, for any array, `list`,

```
Arrays.copyOf( list, lengthOfCopy )
```

is a function that returns a new array whose length is given by `lengthOfCopy`, containing items copied from `list`. If `lengthOfCopy` is greater than `list.length`, then extra spaces in the new array will have their default value (zero for numeric arrays, `null` for object arrays, and so on). If `lengthOfCopy` is less than or equal to `list.length`, then only as many items are copied from `list` as will fit in the new array. So if `A` is any array, then

```
B = Arrays.copyOf( A, A.length );
```

sets `B` to refer to an exact copy of `A`, and

```
playerList = Arrays.copyOf( playerList,
2*playerList.length );
```

could be used to double the amount of space available in a partially full array with just one line of code. We can also use `Arrays.copyOf` to decrease the size of a partially full array. We might want to do that to avoid having a lot of excess, unused spaces. To implement this idea, the code for deleting player number `k` from the list of players might become

```
playerList[k] = playerList[playerCt-1];
playerCt--;
if ( playerCt < playerList.length/4 ) {
        // More than 3/4 of the spaces are empty. Cut the
array size in half.
```

```
    playerList = Arrays.copyOf( playerList,
playerList.length/2 );
}
```

I should mention that class *Arrays* actually contains a bunch of `copyOf` methods, one for each of the primitive types and one for objects. I should also note that when an array of objects is copied, it is only pointers to objects that are copied into the new array. The contents of the objects are not copied. This is the usual rule for assignment of pointers.

If what you want is a simple copy of an array, with the same size as the original, there is an even easier way to do it. Every array has an instance method named `clone()` that makes a copy of the array. To get a copy of an int array, `A`, for example, you can simply say

```
int[] B  =  A.clone();
```

The *Arrays* class contains other useful methods. I'll mention a few of them. As with `Arrays.copyOf`, there are actually multiple versions of all of these methods, for different array types.

- `Arrays.fill( array, value )` — Fill an entire array with a specified value. The type of `value` must be compatible with the base type of the `array`. For example, assuming that `numlist` is an array of type `double[]`, then `Arrays.fill(numlist,17)` will set every element of `numlist` to have the value 17.
- `Arrays.fill( array, fromIndex, toIndex, value )` — Fills part of the `array` with `value`, starting at index number `fromIndex` and ending with index number `toIndex-1`. Note that `toIndex` itself is not included.
- `Arrays.toString( array )` — A function that returns a *String* containing all the values from `array`, separated by commas and enclosed between square brackets. The values in the array are converted into strings in the same way they would be if they were printed out.
- `Arrays.sort( array )` — Sorts the entire array. To sort an array means to rearrange the values in the array so that they are in increasing order. This method works for arrays of *String* and arrays of primitive

type values (except for boolean, which would be kind of silly). But it does not work for all arrays, since it must be meaningful to compare any two values in the array, to see which is "smaller." We will discuss array-sorting algorithms in Section 7.5.

- `Arrays.sort( array, fromIndex, toIndex )` — Sorts just the elements from `array[fromIndex]` up to `array[toIndex-1]`
- `Arrays.binarySearch( array, value )` — Searches for `value` in the `array`. The array **must already be sorted** into increasing order. This is a function that returns an int. If the value is found in the array, the return value is the index of an element that contains that value. If the value does not occur in the array, the return value is -1. We will discuss the binary search algorithm in Section 7.5.

7.2.3 RandomStrings Revisited

One of the examples in Subsection 6.2.4 was a GUI program, RandomStrings.java, that shows multiple copies of a message in random positions, colors, and fonts. When the user clicks a button, the positions, colors, and fonts are changed to new random values. But suppose that we want the message strings to move. That is, we want to run an animation where the strings drift around in the window. In that case, we need to store the properties of each string, since they will be needed to redraw the strings in each frame of the animation. A new version of the program that does that is RandomStringsWithArray.java.

There are 25 strings. We need to store the (x,y) coordinates where each string is drawn, the color of each string, and the font that is used for each string. To make the strings move, I also store a velocity for each string, represented as two numbers `dx` and `dy`. In each frame, the `dx` for each string, multiplied by the elapsed time since the previous frame, is added to the `x` coordinate of the string, similarly for `dy`. Now, the data for the string could be stored in six arrays

```
double[] x = new double[25];
double[] y = new double[25];
double[] dx = new double[25];
double[] dy = new double[25];
```

```
Color[] color = new Color[25];
Font[] font = new Font[25];
```

These arrays would be filled with random values. In the method that draws the strings, the `i`-th 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]`. (The `dx` and `dy` would not be used for the drawing; they are used when updating the data for the next frame.) This would be accomplished by code such as:

```
g.setFill(Color.WHITE); // (Fill with white background.)
g.fillRect(0,0,canvas.getWidth(),canvas.getHeight());
for (int i = 0; i < 25; i++) {
   g.setFill( color[i] );
   g.setFont( font[i] );
   g.fillText( MESSAGE, x[i], y[i] );
   g.setStroke(Color.BLACK);
   g.strokeText( 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 program, I made a simple class to represent all the data that is needed for one copy of the message:

```
private static class StringData {  // Info needed to draw
one string.
    double x,y;        // location of the string;
    double dx,dy;      // velocity of the string;
    Color color;       // color of the string;
    Font font;         // the font that is used to draw the
string
}
```

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. The array has to be created and filled with data. Furthermore, each element of the array is an object of type *StringData* which has to be created before it can be used. The following subroutine creates the array and fills it with random data:

```
private void createStringData() {
    stringData = new StringData[25];
    for (int i = 0; i < 25; i++) {
        stringData[i] = new StringData();
        stringData[i].x = canvas.getWidth() *
Math.random();
        stringData[i].y = canvas.getHeight() *
Math.random();
        stringData[i].dx = 1 + 3*Math.random();
        if (Math.random() < 0.5) // 50% chance that dx is
negative
            stringData[i].dx = -stringData[i].dx;
        stringData[i].dy = 1 + 3*Math.random();
        if (Math.random() < 0.5) // 50% chance that dy is
negative
            stringData[i].dy = -stringData[i].dy;
        stringData[i].color = Color.hsb(
360*Math.random(), 1.0, 1.0 );
        stringData[i].font = fonts[ (int)(5*Math.random())
];
    }
}
```

This method is called in the `start()` method. It is also called to make new random data when the user clicks a button. The strings can now be drawn using a for loop such as:

```
for (int i = 0; i < 25; i++) {
    g.setFill( stringData[i].color );
    g.setFont( stringData[i].font );
    g.fillText( MESSAGE, stringData[i].x, stringData[i].y
);
```

```
    g.setStroke(Color.BLACK);
    g.strokeText( MESSAGE, stringData[i].x,
stringData[i].y );
}
```

But in fact, in my program, I used an equivalent for-each loop, which might be easier to understand:

```
for ( StringData data : stringData ) {
    g.setFill( data.color );
    g.setFont( data.font);
    g.fillText( MESSAGE, data.x, data.y );
    g.setStroke( Color.BLACK );
    g.strokeText( 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.

As for how the animation is done, you can check out the full source code. Animation was discussed in Subsection 6.3.5.

`RandomStringsWithArray` 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, 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.

fontNum = (int)(Math.random() * 5);
switch (fontNum) {
    case 0 -> randomFont = font1;
    case 1 -> randomFont = font2;
    case 2 -> randomFont = font3;
    case 3 -> randomFont = font4;
```

```
    case 4 -> randomFont = font5;
}
```

In the new version of the program, the five fonts are stored in an array, which is named `fonts`. This array is declared as a static final member variable of type `Font[]`:

```
private static final fonts= new Font[] {
        Font.font("Times New Roman", FontWeight.BOLD, 20),
        Font.font("Arial", FontWeight.BOLD, FontPosture
.ITALIC, 28),
        Font.font("Verdana", 32),
        Font.font(40),
        Font.font("Times New Roman", FontWeight.BOLD,
FontPosture .ITALIC, 60)
    };
```

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 ];
```

In fact, the preceding four lines can be replaced by the single line

```
Font randomFont = fonts[ (int)(Math.random() * 5) ];
```

The `switch` statement has been replaced by a single line of code. This is a very typical application of arrays. Note that this example uses the random access property: 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 very easily 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!

7.2.4 Dynamic Arrays

Earlier, we discussed how a partially full array can be used to store a list of players in a game, allowing the list to grow and shrink over the course of the game. The list is "dynamic" in the sense that its size changes while the program is running. Dynamic lists are very common, and we might think about trying to write a class to represent the concept. By writing a class, we can avoid having to repeat the same code every time we want to use a similar data structure. We want something that is like an array, except that its size can change. Think about operations that we might want to perform on a dynamic array. Some essential and useful operations would include

- add an item to the end of the array
- remove the item at a specified position in the array
- get the value of one of the elements in the array
- set the value of one of the elements in the array
- get the number of items currently in the array

When we design our class, these operations will become instance methods in that class. The items in the dynamic array will actually be stored in a normal array, using the partially full array pattern. Using what we know, the class is not difficult to write. We do have to decide what to do when an attempt is made to access an array element that doesn't exist. It seems natural to throw an index-out-of-bounds exception in that case. Let's suppose that the items in the array will be of type int.

```
import java.util.Arrays;

/**
 * Represents a list of int values that can grow and
shrink.
 */
public class DynamicArrayOfInt {

    private int[] items = new int[8];  // partially full
array holding the ints
    private int itemCt;

    /**
     * Return the item at a given index in the array.
     * Throws ArrayIndexOutOfBoundsException if the index
is not valid.
     */
    public int get( int index ) {
        if ( index < 0 || index >= itemCt )
            throw new
ArrayIndexOutOfBoundsException("Illegal index, " + index);
        return items[index];
    }

    /**
     * Set the value of the array element at a given
index.
     * Throws ArrayIndexOutOfBoundsException if the index
is not valid.
     */
    public void set( int index, int item ) {
        if ( index < 0 || index >= itemCt )
            throw new
ArrayIndexOutOfBoundsException("Illegal index, " + index);
        items[index] = item;
    }

    /**
     * Returns the number of items currently in the array.
     */
    public int size() {
        return itemCt;
    }

    /**
     * Adds a new item to the end of the array.  The size
increases by one.
     */
```

```
    public void add(int item) {
        if (itemCt == items.length)
            items = Arrays.copyOf( items, 2*items.length
);
        items[itemCt] = item;
        itemCt++;
    }

    /**
     * Removes the item at a given index in the array.
The size of the array
     * decreases by one.  Items following the removed item
are moved up one
     * space in the array.
     * Throws ArrayIndexOutOfBoundsException if the index
is not valid.
     */
    public void remove(int index) {
        if ( index < 0 || index >= itemCt )
            throw new
ArrayIndexOutOfBoundsException("Illegal index, " + index);
        for (int j = index+1; j < itemCt; j++)
            items[j-1] = items[j];
        itemCt--;
    }

} // end class DynamicArrayOfInt
```

Everything here should be clear, except possibly why the original size of the `items` array is 8. In fact, the number 8 is arbitrary and has no effect on the functionality of the class. Any positive integer would work, but it doesn't make sense for the array to start off very big. The array will grow as needed if the number of items turns out to be large.

The example ReverseInputNumbers.java used a partially full array of int to print a list of input numbers in the reverse of the order in which they are input. In that program, an ordinary array of length 100 was used to hold the numbers. In any given run of the program, the size of the array could be much too large, or it could be too small, resulting in an exception. The program can now be written using a DynamicArrayOfInt, which will adapt itself to any reasonable number of inputs. For the program, see ReverseWithDynamicArray.java. It's a silly program, but the principle holds in any application where the amount of data cannot be predicted in advance:

The size of a dynamic data structure can adapt itself to any amount of data, limited only by the amount of memory available to the program.

This is a nice example, but there is a real problem with it. Suppose that we want to have a dynamic array of *String*. We can't use a *DynamicArrayOfInt* object to hold strings, so it looks like we need to write a whole new class, *DynamicArrayOfString*. If we want a dynamic array to store players in a game, we would need a class *DynamicArrayOfPlayer*. And so on. It looks like we have to write a dynamic array class for every possible type of data! That can't be right! In fact, Java has a solution to this problem, a standard class that implements dynamic arrays and can work with any type of data. The class is called *ArrayList*, and we'll see how it works in the next section.

Section 7.3

ArrayList

AS WE HAVE JUST SEEN in Subsection 7.2.4, we can easily encode the dynamic array pattern into a class, but it looks like we need a different class for each data type. In fact, Java has a feature called "parameterized types" that makes it possible to avoid the multitude of classes, and Java has a single class named *ArrayList* that implements the dynamic array pattern for all data types that are defined as classes (but not, directly, for primitive types).

7.3.1 ArrayList and Parameterized Types

Java has a standard type with the rather odd name `ArrayList<String>` that represents dynamic arrays of *Strings*. Similarly, there is a type `ArrayList<Color>` that can be used to represent dynamic arrays of *Colors*. And if *Player* is a class representing players in a game, then the type `ArrayList<Player>` can be used to represent a dynamic array of *Players*.

It might look like we still have a multitude of classes here, but in fact there is only one class, the *ArrayList* class, defined in the package `java.util`. But *ArrayList* is a **parameterized type**. A parameterized type can take a type parameter, so that from the single class *ArrayList*, we get a multitude of types including `ArrayList<String>`, `ArrayList<Color>`, and in fact `ArrayList<T>` for any object type *T*. The type parameter *T* must be an object type such as a class name or an interface name. It cannot be a primitive type. This means that, unfortunately, you can not have an ArrayList of int or an ArrayList of char.

Consider the type `ArrayList<String>`. As a type, it can be used to declare variables, such as

```
ArrayList<String> namelist;
```

It can also be used as the type of a formal parameter in a subroutine definition, or as the return type of a subroutine. It can be used with the `new` operator to create objects:

```
namelist = new ArrayList<String>();
```

The object created in this way is of type `ArrayList<String>` and represents a dynamic list of strings. It has instance methods such as `namelist.add(str)` for adding a *String* to the list, `namelist.get(i)` for getting the string at index `i`, and `namelist.size()` for getting the number of items currently in the list.

But we can also use *ArrayList* with other types. If *Player* is a class representing players in a game, we can create a list of players with

```
ArrayList<Player> playerList = new ArrayList<Player>();
```

Then to add a player, `plr`, to the game, we just have to say `playerList.add(plr)`. And we can remove player number `k` with `playerList.remove(k)`.

Furthermore if `playerList` is a local variable, then its declaration can be abbreviated to

```
var playlerList = new ArrayList<Player>();
```

using the alternative declaration syntax that was covered in Subsection 4.8.2. The Java compiler uses the initial value that is assigned to `playerList` to deduce that its type is `ArrayList<Player>`.

When you use a type such as `ArrayList<T>`, the compiler will ensure that only objects of type *T* can be added to the list. An attempt to add an object that is not of type *T* will be a syntax error, and the program will not compile. However, note that objects belonging to a subclass of *T* can be added to the list, since objects belonging to a subclass of *T* are still considered to be of type *T*. Thus, if class *Shape* has subclasses *Rectangle*, *Oval* and *RounRect*, then a variable of type `ArrayList<Shape>` can be used to hold objects of type *Rectangle*, *Oval*, and *RoundRect*. (Of course, this is the same way arrays work: An array of type `T[]` can hold objects belonging to any subclass

of *T*.) Similarly, if *T* is an interface, then any object that implements interface *T* can be added to the list.

An object of type `ArrayList<T>` has all of the instance methods that you would expect in a dynamic array implementation. Here are some of the most useful. Suppose that `list` is a variable of type `ArrayList<T>`. Then we have:

- `list.size()` — This function returns the current size of the list, that is, the number of items currently in the list. 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<T>()` creates a list 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 type *T*, or it can be `null`.
- `list.get(N)` — This function returns the value stored at position `N` in the list. The return type of this function is *T*. `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 parameter `obj` must be of type *T*. 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.clear()` — Removes all items from the list, setting its size to zero.
- `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 up one position. The size of the list decreases by 1. This method returns the removed item.
- `list.remove(obj)` — If the specified object occurs somewhere in the list, it is removed from the list. Any items in the list that come after the removed item are moved up 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. If `obj` does not occur in the list, nothing happens; this is not an error. This method returns a boolean value that says whether or not an item was actually removed.

- `list.indexOf(obj)` — A function that searches for the object, `obj`, in the list. If the object is found in the list, then the first position number where it is found is returned. If the object is not found, then `-1` is returned.

For the last two methods listed here, `obj` is compared to an item in the list by calling `obj.equals(item)`, unless `obj` is null. This means, for example, that strings are tested for equality by checking the contents of the strings, not their location in memory.

Java comes with several parameterized classes representing different data structures. Those classes make up the **Java Collection Framework**. Here we consider only *ArrayList*, but we will return to this important topic in much more detail in Chapter 10.

By the way, *ArrayList* can also be used as a non-parametrized type. This means that you can declare variables and create objects of type *ArrayList* such as

```
ArrayList list = new ArrayList();
```

The effect of this is similar to declaring `list` to be of type `ArrayList<Object>`. That is, `list` can hold any object that belongs to a subclass of *Object*. Since every class is a subclass of *Object*, this means that **any** object can be stored in `list`.

7.3.2 Wrapper Classes

As I have already noted, parameterized types don't work with the primitive types. There is no such thing as "`ArrayList<int>`". However, this limitation turns out not to be very limiting after all, because of the so-called **wrapper classes** such as *Integer* and *Character*.

We have already briefly encountered the classes *Double* and *Integer* in Section 2.5. These classes contain the `static` methods `Double.parseDouble()` and `Integer.parseInteger()` that are used to convert strings to numerical values, and they contain constants such as `Integer.MAX_VALUE` and `Double.NaN`. We have also encountered the *Character* class in some examples, with the static method `Character.isLetter()`, that 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 wrapper classes. Although they contain useful `static` members, they have another use as well: They are used for representing primitive type values as objects.

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. This is true, for example, when you would like to store primitive type values in an *ArrayList*. 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 *Integer* contains a single instance variable, of type int. The object is a "wrapper" for the int value. You can get an object that wraps the int value `42` with

```
Integer n = Integer.valueOf(42);
```

The value of `n` has the same information as the value of type int, but it is an object. If you want to retrieve the int value that is wrapped in the object, you can call the function `n.intValue()`. Similarly, you can wrap a double in an object of type *Double*, a boolean value in an object of type *Boolean*, and so on. Each wrapper class has a static `valueOf`() method for wrapping a primitive type value in an object.

The method `Integer.valueOf()` is a `static` factory method that returns an object of type *Integer*. The *Integer* class also has a constructor for creating objects, but it has been **deprecated**, meaning that it should not be used in new code and might be removed from the language in the future. The static factory method has the advantage that if `Integer.valueOf()` is called more than once with the same parameter value, it has the option of returning the

same object each time. This is OK because objects of type *Integer* are **immutable**, that is, the content of the object cannot be modified after the object has been created. Someone who gets their hands on an *Integer* will not be able to change the primitive int value that it represents. We saw something similar for the *Color* class in Subsection 6.2.1, which also has static factory methods for creating immutable objects.

To make the wrapper classes even easier to use, there is 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. If you say

```
Integer answer = 42;
```

the computer will silently read this as if it were

```
Integer answer = Integer.valueOf(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*, and vice versa. In fact, autoboxing and unboxing make it possible in many circumstances to ignore the difference between primitive types and objects.

This is true in particular for parameterized types. Although there is no such thing as "`ArrayList<int>`", there is `ArrayList<Integer>`. An `ArrayList<Integer>` holds objects of type *Integer*, but any object of type *Integer* really just represents an int value in a rather thin wrapper. Suppose that we have an object of type `ArrayList<Integer>`:

```
ArrayList<Integer> integerList;
integerList = new ArrayList<Integer>();
```

Then we can, for example, add an object to `integerList` that represents the number 42:

```
integerList.add( Integer.valueOf(42) );
```

but because of autoboxing, we can actually say

```
integerList.add( 42 );
```

and the compiler will automatically wrap 42 in an object of type *Integer* before adding it to the list. Similarly, we can say

```
int num = integerList.get(3);
```

The value returned by `integerList.get(3)` is of type *Integer* but because of unboxing, the compiler will automatically convert the return value into an int, as if we had said

```
int num = integerList.get(3).intValue();
```

So, in effect, we can pretty much use `integerList` as if it were a dynamic array of int rather than a dynamic array of *Integer*. Of course, a similar statement holds for lists of other wrapper classes such as `ArrayList<Double>` and `ArrayList<Character>`.

There is one issue that sometimes causes problems: A list can hold `null` values, and a `null` does not correspond to any primitive type value. This means, for example, that the statement "`int num = integerList.get(3);`" can produce a null pointer exception in the case where `integerList.get(3)` returns `null`. Unless you are sure that all the values in your list are non-null, you need to take this possibility into account.

7.3.3 Programming With ArrayList

As a simple first example, we can redo ReverseWithDynamicArray.java, from the previous section, using an ArrayList instead of a custom dynamic array class. In this case, we want to store integers in the list, so we should use `ArrayList<Integer>`. Here is the complete program:

```
import textio.TextIO;
import java.util.ArrayList;

/**
 * Reads a list of non-zero numbers from the user, then
prints
 * out the input numbers in the reverse of the order in
which
 * the were entered.  There is no limit on the number of
inputs.
 */
public class ReverseWithArrayList {

    public static void main(String[] args) {
        ArrayList<Integer> list;
        list = new ArrayList<Integer>();
        System.out.println("Enter some non-zero integers.
Enter 0 to end.");
        while (true) {
            System.out.print("? ");
            int number = TextIO.getlnInt();
            if (number == 0)
                break;
            list.add(number);
        }
        System.out.println();
        System.out.println("Your numbers in reverse
are:");
        for (int i = list.size() - 1; i >= 0; i--) {
            System.out.printf("%10d%n", list.get(i));
        }
    }

}
```

As illustrated in this example, ArrayLists are commonly processed using `for` loops, in much the same way that arrays are processed. for example, the following loop prints out all the items for a variable `namelist` of type `ArrayList<String>`:

```
for ( int i = 0; i < namelist.size(); i++ ) {
    String item = namelist.get(i);
    System.out.println(item);
}
```

You can also use for-each loops with ArrayLists, so this example could also be written

```
for ( String item : namelist ) {
    System.out.println(item);
}
```

When working with wrapper classes, the loop control variable in the for-each loop can be a primitive type variable. This works because of unboxing. For example, if `numbers` is of type `ArrayList<Double>`, then the following loop can be used to add up all the values in the list:

```
double sum = 0;
for ( double num : numbers ) {
   sum = sum + num;
}
```

This will work as long as none of the items in the list are `null`. If there is a possibility of null values, then you will want to use a loop control variable of type *Double* and test for nulls. For example, to add up all the non-null values in the list:

```
double sum;
for ( Double num : numbers ) {
    if ( num != null ) {
        sum = sum + num;  // Here, num is SAFELY unboxed
to get a double.
    }
}
```

For a more complete and useful example, we will look at the program SimplePaint2.java. This is a much improved version of SimplePaint.java from Subsection 6.3.3. In the new program, the user can sketch curves in a drawing area by clicking and dragging with the mouse. 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. (Symmetry is there just to look pretty.)

Unlike the original SimplePaint program, this new version uses a data structure to store information about the picture that has been drawn by the user. When the user selects a new background color, the canvas is filled with the new background color, and all of the curves that were there previously are redrawn on the new background. To do that, we need to store enough data to redraw all of the curves. Similarly, the Undo command is implemented by deleting the data for most recently drawn curve, and then redrawing the entire picture using the remaining data.

The data structure that we need is implemented using *ArrayLists*. The main data for an individual curve consists of a list of the points on the curve. This data is stored in an object of type `ArrayList<Point2D>`. (*Point2D* is standard class in package `javafx.geometry`: A *Point2D* can be constructed from two double values, giving the (x,y) coordinates of the point. And a *Point2D* object, `pt`, has getter methods `pt.getX()` and `pt.getY()` that return the x and y coordinates.) But in addition to a list of points on a curve, 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 is grouped into an object of type *CurveData* that is defined as a nested class in the program:

```
private static class CurveData {
   Color color;  // The color of the curve.
   boolean symmetric;  // Are horizontal and vertical
reflections also drawn?
   ArrayList<Point2D> 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, the program uses an *ArrayList*, `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 to represent that curve. The instance variables in the new *CurveData* object must also be initialized. Here is the code from the `mousePressed()` routine that does this, where `currentCurve` is a global variable of type `CurveData`:

```
currentCurve = new CurveData();        // Create a new
CurveData object.

currentCurve.color = currentColor;     // The color of a
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<Point2D>();  // A new
point list object.
```

As the user drags the mouse, new points are added to `currentCurve`, and line segments of the curve are drawn between points as they are added. When the user releases the mouse, the curve is complete, and it is added to the list of curves by calling

```
curves.add( currentCurve );
```

When the user changes the background color or selects the "Undo" command, the picture has to be redrawn. The program has a `redraw()` method that completely redraws the picture. That method uses the data in the list of *CurveData* 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<Point2D>` 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 *Point2D* which has getter methods named `getX()` and `getY()`. We can refer directly to the x-coordinate of the `i-th` point as:

```
curve.points.get(i).getX()
```

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 x coordinate by calling its `getX()` method. Here is the complete definition of the method that redraws the picture:

```
private void redraw() {
    g.setFill(backgroundColor);
    g.fillRect(0,0,canvas.getWidth(),canvas.getHeight());
    for ( CurveData curve : curves ) {
        g.setStroke(curve.color);
        for (int i = 1; i < curve.points.size(); i++) {
                // Draw a line segment from point number
i-1 to point number i.
            double x1 = curve.points.get(i-1).getX();
            double y1 = curve.points.get(i-1).getY();
            double x2 = curve.points.get(i).getX();
            double y2 = curve.points.get(i).getY();
            drawSegment(curve.symmetric,x1,y1,x2,y2);
        }
    }
}
```

`drawSegment()` is a method that strokes the line segment from `(x1,y1)` to `(x2,y2)`. If the first parameter is `true`, it also draws the horizontal and vertical reflections of that segment.

I have mostly been interested here in discussing how the program uses *ArrayList*, but I encourage you to read the full source code, SimplePaint2.java, and to try out the program. In addition to serving as an

example of using parameterized types, it also serves as another example of creating and using menus. You should be able to understand the entire program.

Section 7.4

Records

SOME PROGRAMMING LANGUAGES have two basic kinds of built-in data structures: arrays and records. An array consists of a sequence of items, where individual items are referred to by their numerical position in the sequence. In a record, on the other hand, the positions in the data structure have names instead of numbers. The items in a record are called its "fields," and the names for the items are "field names." A field is accessed using its field name. We recognize records as similar to objects, with the fields in a record playing the same role as instance variables in an object, but records existed before object-oriented programming. The actual word "record" is used in programming languages such as Pascal and Cobol. The C programming language uses the term "struct" for the same idea. The "record" terminology might have originated with databases, which are just large, organized collections of data, where a record is a (typically small) set of related data items, and a database is a collection of records.

In Java, classes can be used to represent records, but the term "record" has not traditionally been used. However, in Java 17, records have become an official part of the language in the form of a special kind of class. Java records are not really equivalent to records in other languages, since a record in Java is immutable, that is, its content cannot be modified after the record is created. However, they are similar to other records is that they are fairly simple containers for named fields.

7.4.1 Basic Java Records

A Java record is an object that belongs to a special kind of class, which I will call a record class. In the simplest case, a record class specifies nothing but the set of instance variables that represent the fields of the record. Here is an example:

```
public record FullName(String firstName, String lastName)
{ }
```

This is a class definition for a record class named *FullName* that has two instance variables of type *String* named `firstName` and `lastName`. These instance variables are the `fields` of the record. The "{ }" at the end of the definition is an empty class body. Note that the instance variables in a record class are listed in parentheses after the class name. The syntax is the same as the syntax for a list of formal parameters in a method definition, but the meaning is very different. A record of type *FullName*—that is, an instance of the record class—is created in the usual way, with the new operator. For example,

```
FullName fname = new FullName("Jane", "Doe");
```

This statement calls a constructor that has one parameter for each field of the record, whose effect is simply to provide a value for each field. Note that this constructor was not explicitly defined in the class. It is called the **canonical constructor** for the record class, and it is provided automatically by the compiler. In fact, many things are added implicitly to a record class definition by the compiler. The simple record definition of *FullName*, given above, is essentially equivalent to the following regular class definition:

```
public final class FullName {
    private final String firstName;
    private final String lastName;
    public FullName( String firstName, String lastName ) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    public firstName() {
        return firstName;
    }
    public lastName() {
        return lastName;
    }
    public String toString() {
       return "FullName[firstName=" + firstName
                       + ", lastName=" + lastName + "]";
    }
    public boolean equals(Object obj) {
        // (definition omitted)
    }
```

```
    public int hashCode() {
        // (definition omitted)
    }
}
```

We see that a record class is automatically `final`, that is, it cannot be extended by subclasses. Furthermore, a record class cannot extend another class (but it is a subclass of *Object*, as is true for any class).

The instance variables in a record are `private` and `final`. Accessor, or "getter", methods for the instance variables are automatically defined, but instead of using the typical `getXXX()` naming convention for getter methods, their names are the same as the names of the instance variables. For example, if `fname` is a variable of type *FullName*, then the instance variables would be accessed as `fname.firstName()` and `fname.lastName()`. Because its instance variables are `final`, a record is immutable, so no "setter" methods can be defined.

Furthermore, reasonable definitions are automatically provided for three methods inherited from class *Object*: `toString()`, `equals()`, and `hashCode`(). The `toString()` method returns a string that includes the name of the class and the names and values of its fields. The `equals()` method returns `true` if its parameter is an object of the same type that has the same values for its fields. (We will not encounter the `hashCode()` method until Section 10.3.)

We will see that record class definitions can be more complex, but you should expect basic record classes, with empty class bodies, to be very common, since they provide a simple way to group together a set of related data items. For example, the *CurveData* class from the SimplePaint2.java example in the previous section was created to group together all the data relevant to a single curve:

```
private static class CurveData {
    Color color;
    boolean symmetric;
    ArrayList<Point2D> points;
}
```

This nested class could be replaced by a nested record class:

```
private record CurveData(
    Color color,
    boolean symmetric,
    ArrayList<Point2D> points
) { }
```

Note that the nested *CurveData* record class does not have to be declared `static`, because nested record classes are automatically `static`.

After this change, when a *CurveData* object is created, values must be provided for its fields. For example,

```
currentCurve = new CurveData(currentColor, useSymmetry,
new ArrayList<Point2D>());
```

Another change is that *CurveData* objects are now immutable. That happens to be OK in `SimplePaint2.java`, but it's not something that will work in all cases. For example, class *Point2D* is a simple container for `xy` coordinates, but it could not be a record class because points are not immutable.

7.4.2 More Record Syntax

It is not possible to add additional instance variables to a record class, beyond those that are defined in the list after the class name. But almost anything else can be added in the class body. For example, a record class can contain `static` items of any kind. It can define instance methods, including replacements for the default `toString()`, `equals()`, and `hashCode()` methods. And it can define constructors, with just a few restrictions. First of all, it is possible to extend the definition for the canonical constructor that is defined automatically, using a syntax in which the parameter list for the constructor is omitted. For example, the canonical constructor for the *FullName* class might throw an exception if `firstName` is `null`:

```
public FullName { // canonical constructor for the
FullName class
    if ( firstName == null) {
        throw new IllegalArgumentException("First name
can't be null.");
    }
}
```

This extends the canonical constructor. Although the parameter list is omitted in the definition, a call to this constructor still requires two parameters, and it still uses those parameters to initialize `firstName` and `lastName`, before the code in the constructor definition is executed.

Additional constructors can be defined, but any non-canonical constructor must begin with a call to a constructor in the same class, using the special variable `this` as discussed in Subsection 5.6.3. This means that the canonical constructor will be called, directly or indirectly, by any other constructor.

As an example, noting that there are people who use only a single name, we might want to provide a *FullName* constructor that takes just one parameter representing that name:

```
public FullName(String onlyName) {
    this( onlyName, null ); // call the canonical
constructor
}
```

This constructor calls the default constructor to set the `firstName` field equal to `onlyName` and the `lastName` field equal to `null`.

We might also want to define a more natural version of `toString()` for the *FullName* record class. For a full class definition that implements all of these ideas, see the sample file FullName.java.

A final syntax note: Although a record class cannot extend another class, it can implement one or more interfaces.

7.4.3 A Complex Example

A **complex number** consists of two real numbers, which are called the real part and the imaginary part of the complex number. Without knowing anything about the mathematics of complex numbers, you should see that this is a natural application for records. To represent a complex number in Java, we need a simple container for two values of type double. That could be done with a basic record class

```
record Complex( double re, double im ) { }
```

But there are many things that can be done with complex numbers, and we would want to include some of those things in a class that could truly be said to represent them. Here is a record class that includes a few of those things:

```
/**
 * A record type for representing complex numbers, where
 * a complex number consists of two real numbers called
its
 * real and imaginary parts.  The class includes methods
for
 * doing arithmetic with complex numbers.
 */
public record Complex(double re, double im)  {

    // Some named constants for common complex numbers.

    public final static Complex ONE = new Complex(1,0);
    public final static Complex ZERO = new Complex(0,0);
    public final static Complex I = new Complex(0,1);

    /**
     * This constructor creates a complex number with a
given
     * real part and with imaginary part zero.
     */
    public Complex(double re) {
        this(re,0);
    }

    /**
     * Creates string representations of complex number
such
     * as:  3.0 + I*5.0,  -I*3.14,   2.7 - I*8.6,   3.14
     */
    public String toString() {
        if (im == 0)
            return String.valueOf(re);
        else if (re == 0) {
            if (im < 0)
                return "-I*" + (-im);
            else
                return "I*" + im;
        }
        else if (im < 0)
```

```
            return re + " - " + "I*" + (-im);
        else
            return re + " + " + "I*" + im;
    }

    // Some methods for doing arithmetic on two complex
numbers

    public Complex plus(Complex c) {
        return new Complex(re + c.re, im + c.im);
    }
    public Complex minus(Complex c) {
        return new Complex(re - c.re, im - c.im);
    }
    public Complex times(Complex c) {
        return new Complex(re*c.re - im*c.im,
                re*c.im + im*c.re);
    }
    public Complex dividedBy(Complex c) {
        double denom = c.re*c.re + c.im*c.im;
        double real = (re*c.re + im*c.im)/denom;
        double imaginary = (im*c.re - re*c.im)/denom;
        return new Complex(real,imaginary);
    }

} // end record Complex
```

This class adds some static member variables, a constructor that creates a complex number from a single real number, a `toString()` method that prints a complex number in a reasonable format, and some instance variables that implement arithmetic operations on complex numbers. Of course, it also has the canonical constructor that creates a complex number from two real numbers. The sample program RecordDemo.java tests both Complex.java and FullName.java.

It might be worth thinking about why record classes should be `final` and why records should be immutable. One reason for them to be `final` is that it can make it possible for a complier to apply certain kinds of optimization to the code that it generates. This applies not just to record classes but to final classes in general.

Here is an example. It is common to compute complicated arithmetic expressions involving complex numbers. Consider the quadratic formula Ax^2+Bx+C. If A, B, C, and x are objects of type *Complex*, then the value of this expression can be computed as

```
(A.times(x).times(x)).plus(B.times(x)).plus(C)
```

If you check the definitions of the `times()` and `plus()` methods, you can see that every time a method is called, it creates a new *Complex* object. In the quadratic formula example, five new objects are generated, but we are only interested in the one that represents the final answer. The other four objects are just scratch work: They are created, used very briefly and then immediately become eligible for garbage collection. Creating and garbage collecting large numbers of objects can be inefficient. However, in this case, the compiler might be able to avoid creating those extra objects by replacing the calls to `plus()` and `times()` with code that performs the same operations directly using temporary local variables of type double instead of objects. It can do this because it knows exactly what each method call does —but that is only the case because the *Complex* class is final. If that were not the case, then A, B, C, or x might actually refer to objects belonging to subclasses of *Complex* that have redefined `plus()` and `times()`. That is something that can only be checked at run time, not at compile time, so if the class were not final, a compiler would have no way of knowing what the calls to `plus()` and `times()` will do when the program is run.

As for immutability, it might also help with optimization, since a compiler can be sure that calling a method on an object will not modify the instance variables in that object. However, it is probably more important that immutability makes it easier to reason about a program. If you can prove that some property of an immutable object is true at some point in time, you can be sure that it won't become false later because the object has been modified.

Section 7.5

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). We have seen in Subsection 7.2.2 that Java has some built-in methods for searching and sorting arrays. However, a computer science student should be familiar with the algorithms that are used in those methods. In this section, you will learn some algorithms for searching and sorting.

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 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 the objects. In that case, we might use the terminology of "records" and "fields" that originated in work with databases, as discussed in the previous section. In the mailing list example, we might have an array of records where each record contains a first name, last name, street address, state, city, and zip code as fields. 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 objects, to remind you of the more practical applications.

7.5.1 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.

Once we know that 42 can only be in the first half of the array, the obvious next 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 implements this idea:

```
/**
 * 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;

}
```

7.5.2 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. (This is **not** meant to be a realistic way to implement a phone directory!)

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 7.2.4) can be used in order to avoid putting an arbitrary limit on the number of entries that the phone directory can hold. A *PhoneDirectory* class should include instance methods that implement the "get" and "put" operations. Here is one possible simple definition of the class:

```
import java.util.Arrays;

/**
 * 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) {
               data = Arrays.copyOf( data, 2*data.length );
            }
            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 also have be written using an ArrayList (Section 7.3) instead of a dynamic array. And the nested *PhoneEntry* class is a natural candidate to be a record class (Section 7.4). For a version that uses these ideas, see PhoneDirectory2.java.

This phone directory implementation could be improved by using binary search instead of simple linear search in the `find()` 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.

I will mention that association lists are also called "maps," and Java has a standard parameterized type named *Map* that implements association lists for keys and values of any type. The implementation is more efficient than anything you can do with basic arrays. You will encounter this class in Section 10.3.

7.5.3 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 technique 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.
   }
}
```

Here 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:

7.5.4 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

}
```

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 the *Hand* class from that section; note that it uses an *ArrayList*. The source file is Hand.java.)

In the *Hand* class, a hand of playing cards is represented by an `ArrayList<Card>`. The objects stored in the list 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 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<Card> newHand = new ArrayList<Card>();
   while (hand.size() > 0) {
      int pos = 0;  // Position of minimal card.
      Card c = hand.get(0);  // Minimal card.
      for (int i = 1; i < hand.size(); i++) {
         Card c1 = hand.get(i);
         if ( c1.getSuit() < c.getSuit() ||
              (c1.getSuit() == c.getSuit() &&
c1.getValue() < c.getValue()) ) {
            pos = i;  // Update the minimal card and
location.
            c = c1;
         }
      }
      hand.remove(pos);  // Remove card from original
hand.
      newHand.add(c);    // Add card to the new hand.
   }
   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`. For example, you can test whether `str1` precedes or is equal to `str2` by testing

```
if ( str1.compareTo(str2) <= 0 )
```

The definition of "precedes" 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 if the strings consist entirely of uppercase letters.) The method `str1.compareToIgnoreCase(str2)` compares the two strings after converting any characters that they contain to lower case.

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, to the same degree that binary search is faster than linear search. The standard method `Arrays.sort` uses these fast sorting algorithms. I'll discuss one such algorithm in Chapter 9.

7.5.5 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;
   }
}
```

Section 7.6

Two-dimensional Arrays

TWO-DIMENSIONAL ARRAYS were introduced in Subsection 3.8.5, but we haven't done much with them since then. A 2D array has a type such as `int[][]` or `String[][]`, with two pairs of square brackets. The elements of a 2D array are arranged in rows and columns, and the `new` operator for 2D arrays specifies both the number of rows and the number of columns. For example,

```
int[][] A;
A = new int[3][4];
```

This creates a 2D array of int that has 12 elements arranged in 3 rows and 4 columns. Although I haven't mentioned it, there are initializers for 2D arrays. For example, this statement creates the 3-by-4 array that is shown in the picture below:

```
int[][]  A  =  {  {  1,  0, 12, -1 },
                  {  7, -3,  2,  5 },
                  { -5, -2,  2, -9 }
               };
```

An array initializer for a 2D array contains the rows of A, separated by commas and enclosed between braces. Each row, in turn, is a list of values separated by commas and enclosed between braces. There are also 2D array literals with a similar syntax that can be used anywhere, not just in declarations. For example,

```
A  =  new int[][] {  {  1,  0, 12, -1 },
                     {  7, -3,  2,  5 },
                     { -5, -2,  2, -9 }
                  };
```

All of this extends naturally to three-dimensional, four-dimensional, and even higher-dimensional arrays, but they are not used very often in practice.

7.6.1 The Truth About 2D Arrays

But before we go any farther, there is a little surprise. Java does not actually have two-dimensional arrays. In a true 2D array, all the elements of the array occupy a continuous block of memory, but that's not true in Java. The syntax for array types is a clue: For any type *BaseType*, we should be able to form the type `BaseType[]`, meaning "array of *BaseType*." If we use `int[]` as the base type, the type that we get is "`int[][]` meaning "array of `int[]`" or "array of array of int." And in fact, that's what happens. The elements in a 2D array of type `int[][]` are variables of type `int[]`. And remember that a variable of type `int[]` can only hold a pointer to an array of int. So, a 2D array is really an array of pointers, where each pointer can refer to a one-dimensional array. Those one-dimensional arrays are the rows of the 2D array. A picture will help to explain this. Consider the 3-by-4 array `A` defined above.

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[]`.

Some of the consequences of this structure are a little subtle. For example, thinking of a 2D array, `A`, as an array of arrays, we see that `A.length` makes sense and is equal to the number of rows of `A`. If `A` has the usual shape for a 2D array, then the number of columns in `A` would be the same as the number of elements in the first row, that is, `A[0].length`. But there is no rule that says that all of the rows of `A` must have the same length (although an array created with `new BaseType[rows][columns]` will always have that form). Each row in a 2D array is a separate one-dimensional array, and each of those arrays can have a different length. In fact, it's even possible for a row to be `null`. For example, the statement

```
A = new int[3][];
```

with no number in the second set of brackets, creates an array of 3 elements where all the elements are `null`. There are places for three rows, but no actual rows have been created. You can then create the rows `A[0]`, `A[1]`, and `A[2]` individually.

As an example, consider a **symmetric matrix**. A symmetric matrix, `M`, is a two-dimensional array in which the number of rows is equal to the number of columns and satisfying `M[i][j]` equals `M[j][i]` for all `i` and `j`. Because of this equality, we only really need to store `M[i][j]` for `i >= j`. We can store the data in a "triangular matrix":

It's easy enough to make a triangular array, if we create each row separately. To create a 7-by-7 triangular array of double, we can use the code segment

```
double[][] matrix = new double[7][]; // rows have not yet
been created!
for (int i = 0; i < 7; i++) {
    matrix[i] = new double[i+1];  // Create row i with i +
1 elements.
}
```

We just have to remember that if we want to know the value of the matrix at `(i,j)`, and if `i < j`, then we actually have to get the value of `matrix[j][i]` in the triangular matrix. And similarly for setting values. It's easy to write a class to represent symmetric matrices:

```
/**
 * Represents symmetric n-by-n matrices of real numbers.
 */
public class SymmetricMatrix {

    private double[][] matrix;  // A triangular matrix to
hold the data.

    /**
     * Creates an n-by-n symmetric matrix in which all
entries are 0.
     */
    public SymmetricMatrix(int n) {
        matrix = new double[n][];
        for (int i = 0; i < n; i++)
            matrix[i] = new double[i+1];
    }

    /**
     * Returns the matrix entry at position (row,col).
(If row < col,
     * the value is actually stored at position
(col,row).)
     */
    public double get( int row, int col ) {
        if (row >= col)
            return matrix[row][col];
        else
            return matrix[col][row];
    }

    /**
     * Sets the value of the matrix entry at (row,col).
(If row < col,
```

```
     * the value is actually stored at position
(col,row).)
     */
    public void set( int row, int col, double value ) {
        if (row >= col)
            matrix[row][col] = value;
        else
            matrix[col][row] = value;
    }

    /**
     * Returns the number of rows and columns in the
matrix.
     */
    public int size() {
        return matrix.length;  // The size is the number
of rows.
    }

} // end class SymmetricMatrix
```

This class is in the file SymmetricMatrix.java, and a small program to test it can be found in TestSymmetricMatrix.java.

By the way, the standard function `Arrays.copyOf()` can't make a full copy of a 2D array in a single step. To do that, you need to copy each row separately. To make a copy of a two-dimensional array of int, for example:

```
int[][] B = new int[A.length][];  // B has as many rows as
A.
for (int i = 0; i < A.length; i++) {
    B[i] = Arrays.copyOf(A[i], A[i].length)); // Copy row
i.
}
```

7.6.2 Conway's Game Of Life

As an example of more typical 2D array processing, let's look at a very well-known example: Conway's Game of Life, invented by mathematician John Horton Conway in 1970. This Game of Life is not really a game (although sometimes it's referred to as a "zero-person game" that plays itself). It's a "two-dimensional cellular automaton." This just means that it's a grid of cells

whose content changes over time according to definite, deterministic rules. In Life, a cell can only have two possible contents: It can be "alive" or "dead." We will use a 2D array to represent the grid, with each element of the array representing the content of one cell in the grid. In the game, an initial grid is set up in which each cell is marked as either alive or dead. After that, the game "plays itself." The grid evolves through a series of time steps. The contents of the grid at each time step are completely determined by the contents at the previous time step, according to simple rules: Each cell in the grid looks at its eight neighbors (horizontal, vertical, and diagonal) and counts how many of its neighbors are alive. Then the state of the cell in the next step is determined by the rules:

- For a cell that is alive in the current time step: If the cell has 2 or 3 living neighbors, then the cell remains alive in the next time step; otherwise, it dies. (A living cell dies of loneliness if it has 0 or 1 living neighbor, and of overcrowding if it has more than 3 living neighbors.)
- For a cell that is dead in the current time step: If the cell has 3 living neighbors, then the cell becomes alive in the next time step; otherwise, it remains dead. (Three living cells give birth to a new living cell.)

Here's a picture of part of a Life board, showing the same board before and after the rules have been applied. The rules are applied to every cell in the grid. The picture shows how they apply to four of the cells:

The Game of Life is interesting because it gives rise to many interesting and surprising patterns. (Look it up on Wikipedia.) Here, we are just interested in writing a program to simulate the game. The complete program can be found

in the file Life.java. In the program, the life grid is shown as a grid of squares in which dead squares are black and living squares are white. (The program uses MosaicCanvas.java from Section 4.7 to represent the grid, so you will also need that file to compile and run the program.) In the program, you can fill the life board randomly with dead and alive cells, or you can use the mouse to set up the game board. There is a "Step" button that will compute one time-step of the game, and a "Start" button that will run time steps as an animation.

We'll look at some of the array processing involved in implementing the Game of Life for this program. Since a cell can only be alive or dead, it is natural to use a two-dimensional array of `boolean[][]` to represent the states of all the cells. The array is named `alive`, and `alive[r][c]` is true when the cell in row `r`, column `c` is alive. The number of rows and the number of columns are equal and are given by a constant, `GRID_SIZE`. So, for example, to fill the Life grid with random values, the program uses simple nested `for` loops:

```
for (int r = 0; r < GRID_SIZE; r++) {
    for (int c = 0; c < GRID_SIZE; c++) {
            // Use a 25% probability that the cell is
alive.
        alive[r][c] = (Math.random() < 0.25);
    }
}
```

Note that the expression `(Math.random() < 0.25)` is a true/false value that can be assigned to a boolean array element. The array is also used to set the color of the cells on the screen. Since the grid of cells is displayed on screen as a *MosaicCanvas*, setting the colors is done using the MosaicCanvas API. Note that the actual drawing is done in the *MosaicCanvas* class (which has its own 2D array of type `Color[][]` to keep track of the colors of each cell). The Life program just has to set the colors in the mosaic, using the MosaicCanvas API. This is done in the program in a method named `showBoard()` that is called each time the board changes. Again, simple nested `for` loops are used to set the color of each square in the grid:

```
for (int r = 0; r < GRID_SIZE; r++) {
    for (int c = 0; c < GRID_SIZE; c++) {
        if (alive[r][c])
```

```
            display.setColor(r,c,Color.WHITE);
        else
            display.setColor(r,c,null);  // Shows the
background color, black.
    }
}
```

Of course, the most interesting part of the program is computing the new state of the board by applying the rules to the current state. The rules apply to each individual cell, so again we can use nested for loops to work through all the cells on the board, but this time the processing is more complicated. Note first that we can't make changes to the values in the array as we work through it, since we will need to know the old state of a cell when processing its neighboring cells. In fact, the program uses a second array to hold the new board as it is being created. When the new board is finished, it can be substituted for the old board. The algorithm goes like this in pseudocode:

```
let newboard be a new boolean[][] array
for each row r:
    for each column c:
        Let N be the number of neighbors of cell (r,c) in
the alive array
        if ((N is 3) or (N is 2 and alive[r][c]))
            newboard[r][c] = true;
        else
            newboard[r][c] = false;
alive = newboard
```

Note that at the end of the process, `alive` is pointing to a new array. This doesn't matter as long as the contents of the array represent the new state of the game. The old array will be garbage collected. The test for whether `newboard[r][c]` should be `true` or `false` might not be obvious, but it implements the rules correctly. We still need to work on counting the neighbors. Consider the cell in row `r` and column `c`. If it's not at an edge of the board, then it's clear where its neighbors are:

The row above row number `r` is row number `r-1`, and the row below is `r+1`. Similarly for the columns. We just have to look at the values of `alive[r-1][c-1]`, `alive[r-1][c]`, `alive[r-1][c+1]`, `alive[r][c-1]`, `alive[r][c+1]`, `alive[r+1][c-1]`, `alive[r+1][c]`, and `alive[r+1][c+1]`, and count the number that are `true`. (You should make sure that you understand how the array indexing works here.)

But there is a problem when the cell is along one of the edges of the grid. In that case, some of the array elements in the list don't exist, and an attempt to use them will cause an exception. To avoid the exception, we have to give special consideration to cells along the edges. One idea is that before referencing any array element, check that the array element actually exists. In that case, the code for neighbor counting becomes

```
if (r-1 >= 0 && c-1 >= 0 && alive[r-1][c-1])
    N++; // A cell at position (r-1,c-1) exists and is
alive.
if (r-1 >= 0 && alive[r-1][c])
    N++; // A cell at position (r-1,c) exists and is
alive.
if (r-1 >= 0 && c+1 <= GRID_SIZE && alive[r-1][c+1])
    N++; // A cell at position (r-1,c+1) exists and is
alive.
// and so on...
```

All the possible exceptions are avoided. But in my program, I actually do something that is common in 2D computer games—I pretend that the left edge of the board is attached to the right edge and the top edge to the bottom edge. For example, for a cell in row 0, we say that the row "above" is actually the bottom row, row number `GRID_SIZE-1`. I use variables to represent the positions above, below, left, and right of a given cell. The code turns out to

be simpler than the code shown above. Here is the complete method for computing the new board:

```
private void doFrame() { // Compute the new state of the
Life board.
    boolean[][] newboard = new boolean[GRID_SIZE]
[GRID_SIZE];
    for ( int r = 0; r < GRID_SIZE; r++ ) {
        int above, below; // rows considered above and
below row number r
        int left, right;  // columns considered left and
right of column c
        above = r > 0 ? r-1 : GRID_SIZE-1;  // (for "?:"
see Subsection 2.5.5)
        below = r < GRID_SIZE-1 ? r+1 : 0;
        for ( int c = 0; c < GRID_SIZE; c++ ) {
            left =  c > 0 ? c-1 : GRID_SIZE-1;
            right = c < GRID_SIZE-1 ? c+1 : 0;
            int n = 0; // number of alive cells in the 8
neighboring cells
            if (alive[above][left])
                n++;
            if (alive[above][c])
                n++;
            if (alive[above][right])
                n++;
            if (alive[r][left])
                n++;
            if (alive[r][right])
                n++;
            if (alive[below][left])
                n++;
            if (alive[below][c])
                n++;
            if (alive[below][right])
                n++;
            if (n == 3 || (alive[r][c] && n == 2))
                newboard[r][c] = true;
            else
                newboard[r][c] = false;
        }
    }
    alive = newboard;
}
```

Again, I urge you to check out the source code, Life.java, and try the program. Don't forget that you will also need MosaicCanvas.java.

7.6.3 Checkers

As a final example for this chapter, we'll look at a more substantial example of using a 2D array. This is the longest program that we have encountered so far, with 741 lines of code. The program lets two users play checkers against each other. The checkers game is played on an eight-by-eight board, which is based on an example from Subsection 6.5.1. The players are called "red" and "black," after the color of their checkers. I'm not going to explain the rules of checkers here; possibly you can learn them by trying out the program.

In the program, a player moves by clicking on the piece that they want to move, and then clicking on the empty square to which it is to be moved. As an aid to the players, any square that the current player can legally click at a given time is highlighted with a brightly colored border. The square containing a piece that has been selected to be moved, if any, is surrounded by a yellow border. Other pieces that can legally be moved are surrounded by a cyan-colored border. If a piece has already been selected to be moved, 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. Here is a picture of the program early in a game. It is black's turn to move. Black has selected the piece in the yellow-outlined square to be moved. Black can click one of the squares outlined in green to complete the move, or can click one of the squares outlined in cyan to select a different piece to be moved.

I will only cover a part of the programming for this example. I encourage you to read the complete source code, Checkers.java. It's long and complex, but with some study, you should understand all the techniques that it uses. The program is a good example of state-based, event-driven, object-oriented 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 not directly responsible for any part of the graphics or event-handling, but it provides methods that can be called by other classes that handle graphics and events. 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 array 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 can move in both directions.

One function of the *CheckersData* class is to take care of changes to the data structures that need to be made when one of the users moves a checker. An instance method named `makeMove()` is provided to do this. When a player moves a piece from one square to another, the values of two elements 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. Putting all that into a subroutine 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; // The square it moved
from is now 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:  // There are no legal jumps!
   Find any other legal moves and add them to moves
if moves.size() is 0:  // There are no legal moves at all!
   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)
```

```
   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;  // (This will not happen in a correct
program.)

   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
```

The checkers program is complex, and you can be sure that it didn't just fall together. It took a good deal of design work to decide what classes and objects would be used, what methods should be written, and what algorithms the methods should use. The complete source code is in the file Checkers.java. Take a look!

Programming Exercises for Chapter 7

Exercise 7.1:

Write a subroutine that creates an ArrayList containing several *different* random integers in the range from 1 up to some specified maximum. The number of integers and the maximum allowed value for the integers should be parameters to the subroutine. Write a `main()` routine to test your subroutine.

See the Solution

Exercise 7.2:

Suppose that `M` is a two-dimensional array that has `R` rows and `C` columns. The **transpose** of `M` is defined to be an array `T` that has `C` rows and `R` columns such that `T[i][j] = M[j][i]` for each `i` and `j`. Write a function that takes an array of type `int[][]` as a parameter, and returns the transpose of that array. (Assume that the parameter is a typical 2D array in which all the rows have the same length.) Also write a subroutine to print a 2D array of integers in neat rows and columns, and include a `main()` routine to test your work.

See the Solution

Exercise 7.3:

In Subsection 7.5.4, it is mentioned that the standard sorting method `Arrays.sort()` is much faster and efficient than selection sort. Write a program to test this claim. To be specific, your program should create a large array filled with random real numbers. It should use both `Arrays.sort()` and `selectionSort()` to sort the array, and it should time how long it takes to perform each sort. Furthermore, it should do the same thing for a large

array of random *Strings*. To find the times, you can use `System.nanoTime()` (see Subsection 2.3.1 and the example TimedComputation.java).

See the Solution

Exercise 7.4:

In Exercise 6.2, you wrote a program `DragTwoSquares` that allows the user to drag a red square and a blue square around on a canvas. Write a much improved version where the user can add squares to a canvas and drag them around. In particular: If the user shift-clicks or right-clicks the canvas, then the user is trying to drag a square; find the square that contains the mouse position, if any, and move it as the user drags the mouse. Other clicks should add squares. You can place the center of the new square at the current mouse position. To make the picture more visually appealing, give each square a random color, and when you draw the squares, draw a black outline around each square. (My program also gives the square a random alpha value between 0.5 and 1.0).

Write a class to represent the data needed for drawing one square, and use an *ArrayList* to store the data for all the squares in the picture. If the user drags a square completely off the canvas, delete it from the list.

See the Solution

Exercise 7.5:

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. Do **not** use any built-in function such as `Arrays.sort()`. Do the sorting yourself.

See the Solution

Exercise 7.6:

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 2.4.4, you can use *TextIO* to read and write files. Use a variable of type `ArrayList<String>` to store the words. It is not easy to separate a file into words as you are reading it, especially if you want to allow apostrophes in the middle of a word. You can use the following method in your program:

```
/**
 * 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.

See the Solution

Exercise 7.7:

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. The board should have 13 rows and 13 columns of squares. 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 7.6.3. Play alternates strictly between the two players, and there is no need to highlight the legal moves. You will only need one nested subclass, a subclass of *Canvas* to draw the board and do all the work of the game, like the nested *CheckersBoard* class in the *Checkers* program. 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 program. 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 player's 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, just after black has won the game.

See the Solution

Quiz on Chapter 7

Question 1:

What is meant by the *basetype* of an array?

Question 2:

What is the purpose of the following variable-arity method? What are the values of `same(1,2,3)`, `same(17,17,17,17)`, and `same(2)`? Why?

```
static double same( int... value ) {
    for (int i = 1; i < value.length; i++) {
        if ( value[i-1] != value[i] )
            return false;
    }
    return true;
}
```

Question 3:

What does it mean to sort an array?

Question 4:

What is the main advantage of binary search over linear search? What is the main disadvantage?

Question 5:

What is meant by a *dynamic array?* What is the advantage of a dynamic array over a regular array?

Question 6:

What does it mean to say that *ArrayList* is a parameterized type?

Question 7:

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.

Question 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();
}
```

Question 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.

Question 10:

Suppose that temperature measurements were made on each day of 2021 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.

Question 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. Write two versions of the code segment, one that uses a regular `for` loop, and one that uses a for-each loop.

Question 12:

Convert the *Employee* class from the previous question into a record class. What changes would then need to be made to the previous question's solution?

Question 13:

Write a record class to represent dates, where a *Date* object contains three integer fields giving the month, day, and year of the date. The canonical constructor should throw an exception if the values for the month and day are not legal. Also include a `toString()` method that prints a date in a form such as "5/27/2022".

Question 14:

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 double[] sums( double[][] data ) {
    double[] answers = new double[ data.length ];
    for (int i = 0; i < data.length; i++) {
        double sum = 0;
        for (int j = 0; j < data[i].length; i++)
            sum = sum + data[i][j];
        answers[i] = sum;
    }
    return answers;
}
```

See the Answers

Chapter 8

Correctness, Robustness, Efficiency

IN PREVIOUS CHAPTERS, we have covered programming fundamentals. 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, web sites to go offline, 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. Section 8.2 discusses how you can think about and analyze programs to make the programs that you write more likely to be correct—possibly even provably correct.

Section 8.3 looks more closely at exceptions and the `try..catch` statement, and Section 8.4 introduces **assertions**, another of the tools that Java provides as an aid in writing correct programs.

In Section 8.5, we 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 efficiency of algorithms.

Some of the topics in this chapter are the topics of advanced courses in computer science, and only a brief overview can be given here. But what you do learn here will be useful in the rest of the book.

Contents of Chapter 8:

- Section 1: Introduction to Correctness and Robustness
- Section 2: Writing Correct Programs
- Section 3: Exceptions and try..catch
- Section 4: Assertions and Annotations
- Section 5: Analysis of Algorithms
- Programming Exercises
- Quiz on This Chapter

Section 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.

About twenty-three years 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 high above the ground. 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.

Some programming languages do 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 first Macintosh computer, a long time ago, 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. In fact, Java's internal security errors have at times been a problem for the language.)

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);
}
```

(Be sure you understand why we can't just test "`if (3*N+1 > 2147483647)`".) 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 at the start of the year 2000 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.

Section 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 {
   System.out.print("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)`", and if there are no `break` statements in the loop, 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.)

8.2.2 Preconditions and Postconditions

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 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 4.7.1. 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 in general.

Let's see how this works by considering a longer program segment:

```
do {
   System.out.println("Enter A, B, and C.");
   System.out.println("A must be non-zero and B*B-4*A*C
must be >= 0.");
   System.out.print("A = ");
   A = TextIO.getlnDouble();
   System.out.print("B = ");
   B = TextIO.getlnDouble();
   System.out.print("C = ");
   C = TextIO.getlnDouble();
   if (A == 0 || B*B - 4*A*C < 0)
      System.out.println("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 $A*x^2 + B*x + C = 0$ is also valid. This program segment correctly and provably computes a solution to the equation. (Actually, because of problems with representing real 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.

```
System.out.println("Enter your values for A, B, and C.");
System.out.print("A = ");
A = TextIO.getlnDouble();
System.out.print("B = ");
B = TextIO.getlnDouble();
System.out.print("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);
   System.out.println("A solution of A*X*X + B*X + C = 0
is " + x);
}
else if (A == 0) {
   System.out.println("The value of A cannot be zero.");
}
else {
   System.out.println("Since B*B - 4*A*C is less than
zero, the");
   System.out.println("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);
```

8.2.3 Invariants

Let's look at how loops work in more detail. Consider a subroutine for finding the sum of the elements in an array of int:

```
static int arraySum( int[] A ) {
    int total = 0;
    int i = 0;
    while ( i < A.length ) {
        total = total + A[i];
        i = i + 1;
    }
    return total;
}
```

(Note, by the way, that the requirements that `A` is not `null` is a precondition of the subroutine. If it is violated, the code in the subroutine will throw a `NullPointerException`.)

How can we be sure that this subroutine works? We need to prove that when the `return` statement is executed, the value of `total` is the sum of all the elements in `A`. One way to think about this problem is in terms of **loop invariants**.

A loop invariant is, roughly, a statement that remains true as the loop is executed. More precisely, we can show that a statement is an invariant for a loop if the following holds: As long as the statement is true **before** the code inside the loop is executed, then it will also be true **after** the code inside the loop has been executed. That is, a loop invariant is both a precondition and a postcondition of the body of the loop.

A loop invariant for the loop in the above subroutine is, "`total` is equal to the sum of the first `i` elements of `A`." Suppose this is true at the beginning of the `while` loop. That is, before the statement "`total = total + A[i]`" is executed, `total` is the sum of the first `i` elements of the array (namely `A[0]` through `A[i-1]`). After `A[i]` is added to `total`, `total` is now the sum of the

first `i+1` elements of the array. At this point, the loop invariant is **not** true. However, as soon as the next statement, "`i = i + 1`" is executed, replacing `i` with `i+1`, the loop invariant becomes true again. We have checked that if the loop invariant is true at the start of the body of the loop, then is also true at the end.

Note that a loop invariant is not necessarily true at **every** point during the execution of a loop. Executing one of the statements in the loop can make it false temporarily, as long as later statements in the loop make it true again.

So, have we proved that the subroutine `arraySum()` is correct? Not quite. There are still a few things to check. First of all, we need to make sure that the loop invariant is true before the very first time the loop is executed. At that point, `i` is zero, and `total` is also equal to zero, which is the correct sum of zero elements. So the loop invariant is true before the loop. Once we know that, we know that it remains true after each execution of the loop (because it's an invariant), and in particular, we know that it will still be true after the loop ends.

But for that to do us any good, we need to check that the loop actually does end! In each execution of the loop, the value of `i` goes up by one. That means that eventually it has to reach `A.length`. At that point, the condition in the `while` loop is false, and the loop ends.

After the loop ends, we know that `i` equals `A.length`, and we know that the loop invariant is true. At that point, since `i` is `A.length`, the loop invariant says, "`total` is the sum of the first `A.length`" elements of A." But that includes **all** of the elements of `A`. So, the loop invariant gives us exactly what we wanted to show: When `total` is returned by the subroutine, it is equal to the sum of all the elements of the array!

This might seem to you to be a lot of work to prove something that's obvious. But if you try to explain *why* it's obvious that `arraySum()` works, you'll probably find yourself using the logic behind loop invariants, even if you don't use the term.

Let's look more quickly at a similar example. Consider a subroutine that finds the maximum value in an array of int, where we assume that the array has

length at least one:

```
static int maxInArray( int[] A ) {
    int max = A[0];
    int i = 1;
    while ( i < A.length ) {
        if ( A[i] > max )
            max = A[i];
        i = i + 1;
    }
    return max;
}
```

In this case, we have a loop invariant that says, "`max` is the largest value among the first `i` elements of `A`." This statement is true before the loop starts, when `i` is 1 and `max` is `A[0]`. Suppose it is true at the start of the loop, before the `if` statement. After the `if` statement, `max` is greater than or equal to `A[i]`, because that is a postcondition of the `if` statement, and it is greater than or equal to `A[0]` through `A[i-1]`, because of the truth of the loop invariant. Put those two facts together, and you get that `max` is the largest value among the first `i+1` elements of `A`. When `i` is replaced by `i+1` in the next statement, the loop invariant becomes true again. After the loop ends, `i` is `A.length`, and the loop invariant tells us exactly what we need to know: `max` is the largest value in the whole array.

Loop invariants are not just useful for proving that programs are correct. Thinking in terms of loop invariants can be useful when you are trying to develop an algorithm. As an example, let's look at the insertion sort algorithm that was discussed in Subsection 7.5.3. Suppose that we want to sort an array `A`. That is, at the end of the algorithm, we want it to be true that

```
A[0] <= A[1] <= ... <= A[A.length-1]
```

The question is, what step-by-step procedure can we use to make this statement true? Well, can we come up with a loop invariant that, at the end, will become the statement that we want to be true? If we want all of the elements to be sorted at the end, how about a loop invariant that says that some of the elements are sorted—say, that the first `i` elements are sorted. This leads to an outline for the algorithm:

```
i = 0;
while (i < A.length ) {
   // Loop invariant:  A[0] <= A[1] <= ... <= A[i-1]
      .
      .  // Code that adds A[i] to the sorted portion of
the array
      .
   i = i + 1;
}
// At this point, i = A.length, and A[0] <= A[1] <= ... <=
A[A.length-1]
```

The loop invariant is true before the `while` loop, and when the loop ends, the loop invariant becomes precisely the statement that we want to be true at the end of the algorithm. We know what we have to do to complete the algorithm: Develop code for the inside of the loop that will preserve the truth of the loop invariant. If we can do that, the loop invariant will assure us that the algorithm that we have developed is correct. The algorithm for adding `A[i]` to the sorted portion of the array will require its own loop, with its own loop invariant. I'll leave you to think about that.

There is another kind of invariant that is useful for thinking about programs: **class invariants**. A class invariant is, roughly, a statement that is always true about the state of a class, or about objects created from that class. For example, suppose we have a *PairOfDice* class in which the values shown on the dice are stored in instance variables `die1` and `die2`. (See Section 5.2 for a variety of such classes.) We might like to have a class invariant that says, "the values of `die1` and `die2` are in the range 1 through 6." (This would be a statement about any object created from the *PairOfDice* class, not about the class as such.) After all, this is a statement that should always be true about any pair of dice.

But in order to be a class invariant, the statement has to be guaranteed true at all times. If `die1` and `die2` are `public` instance variables, then no such guarantee is possible, since there is no way to control what values a program that uses the class might assign to them. So, we are led to make `die1` and `die2` `private`. Then we just have to make sure that all of the code in the class definition respects the class invariant. That is, first of all, when a *PairOfDice* object is constructed, the variables `die1` and `die2` must be

initialized to be in the range 1 to 6. Furthermore, every method in the class must preserve the truth of the class invariant. In this case, that means that any method that assigns a value to `die1` or `die2` must ensure that the value is in the range 1 to 6. For example, a setter method would have to check that a legal value is being assigned.

In general, we can say that a class invariant is a postcondition of every constructor and is both a precondition and a postcondition of every method in the class. When you are writing a class, a class invariant is something that you *want to be true* at all times. When you write a method, you need to make sure that the code in that method respects the invariant: Assuming that the class invariant is true when the method in called, you need to ensure that it will still be true after the code in the method is executed. This kind of thinking can be a very useful tool for class design.

As another example, consider a dynamic array class, like the one in Subsection 7.2.4. That class uses an ordinary array to store values and a counter to keep track of how many items have been added to the dyanmic array:

```
private int[] items = new int[8];
private int itemCount = 0;
```

Class invariants include the facts that "`itemCount` is the number of items," that "`0 <= itemCount < items.length`," and that "the items are in the array elements `items[0]` through `items[itemCount-1]`." Keeping these invariants in mind can be helpful when writing the class. When writing a method for adding an item, the first invariant reminds you to increment `itemCount` in order to ensure that the invariant remains true. The second invariant tells you where the new item has to be stored. And the third invariant tells you that if incrementing `itemCount` makes it equal to `items.length`, then you will need to do something to avoid violating the invariant. (Since `itemCount` has to be incremented, the invariant means that you will have to make the array bigger.)

In future chapters, I will occasionally point out how it can be useful to think in terms of preconditions, postconditions, and invariants.

I should note that reasoning about invariants becomes much more complicated in parallel programs, when several threads that are running at the same time and are accessing the same data. This will be an issue when we encounter threads in Chapter 12.

8.2.4 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 `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 "1ft". 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: We have to make sure that the next character in the input is in fact either an end-of-line or is a non-blank. To ensure that, we need to skip over any blank characters. 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. The call to `skipBlanks()` ensures that the precondition for the test is again true. 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. You might want to run the program and try this out.

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) ) {
           System.out.println(
                 "Error:  Expected to find a number, but
found " + ch);
           return -1;
       }
       measurement = TextIO.getDouble();

       skipBlanks();
       if (TextIO.peek() == '\n') {
           System.out.println(
                 "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 {
           System.out.println("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.

Section 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 algorithm into a messy tangle of `if` statements.

8.3.1 Exceptions and Exception Classes

We have already seen in Section 3.7 that Java 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 red:

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. Information in the stack trace can tell you exactly where in the program the exception occurred. (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 has many options. It will take a while to go through them. For one thing, a `try..catch` statement can have 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.

Of course, catching *RuntimeException* would catch many more types of exception than the two that we are interested in. It is possible to combine several specific exception types in 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 ( NullPointerException |
ArrayIndexOutOfBoundsException err ) {
   System.out.println("Sorry, an error has occurred.");
   System.out.println("The error was: " + err);
}
```

Here, the two exception types are combined with a "`|`", the vertical line character that is also used in the boolean **or** operator. This example will catch errors of type *NullPointerException* or *ArrayIndexOutOfBoundsException*, and no other types.

The example I've been using here is not realistic, because 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. The next variation is the possibility of a **finally clause** at the end of a `try` statement. With this addition, the 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-names variable-name ) {
   statements
}
```

where **exception-class-names** can be a single exception class or several classes separated by "`|`". 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
   communicate over the connection
}
catch ( IOException e ) {
   report the error
}
finally {
   if the connection was successfully opened
      close the connection
}
```

The `finally` clause ensures that the network connection will definitely be closed, whether or not an error occurs during the communication. 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.

The pattern of obtaining a resource, then using the resource, and then releasing the resource is very common. Note that the resource can only be released if no error occurred while obtaining it. And, if it was successfully obtained, then it should be closed whether or not an error occurs while using it. This pattern is so common that it leads to one last option in the `try` statement syntax. With this option, you only need code to obtain the resource, and you don't need to worry about releasing it. That will happen automatically at the end of the `try` statement.

In order for this to work, the resource must be represented by an object that implements an interface named *AutoCloseable*, which defines a single method named `close()`, with no parameters. Standard Java classes that represent things like files and network connections already implement

AutoClosable. So does the *Scanner* class, which was introduced in Subsection 2.4.6. In that section, I showed how to use a Scanner to read from `System.in`. Although I didn't do it in that section, it's considered good form to close a Scanner after using it. Here is an example that uses the pattern in a `try` statement to make sure that the Scanner is closed automatically:

```
try( Scanner in = new Scanner(System.in) ) {
    // Use the Scanner to read from standard input
}
catch (Exception e) {
    // ... some error occurred while using the Scanner
}
```

The statement that allocates the *Scanner* goes in parentheses after the word "try". The statement must have the form of a variable declaration that includes an initialization of the variable. The variable is local to the `try` statement. (You can actually declare several variables in the parentheses, separated by semicolons.) In this example, we can be sure that `in.close()` will definitely be called by the system at the end of the `try` statement, as long as the Scanner was successfully initialized.

This is all getting quite complicated, and I won't continue the discussion here. The sample program TryStatementDemo.java demonstrates a `try` statement with all its options, and it includes a lot of comments to help you understand what can happen when you run the program.

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 4.3.8. 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. This is called "unwinding the call stack."

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 no caller of the subroutine can ever 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 the 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 checked exceptions 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.");
```

Since *ParseError* is defined as a subclass of *Exception*, it is a checked exception. If the `throw` statement does not occur in a `try` statement that catches the error, then the subroutine that contains the `throw` 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 global 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 red 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.

Section 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. (Annotations, in particular, don't really belong in this chapter, but I could not find a better place to put my short introduction to the topic.)

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.

And the same thing applies to loop invariants and class invariants. These are things that **should** be true at certain points in a program. If they are not true at those points, it means that the program contains a bug.

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.
 * 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 or an invariant. These are conditions that are definitely expected to be true in any bug-free program, so an assertion is the natural way to check the condition while debugging, without imposing an efficiency penalty when the program is executed normally. If the postcondition or invariant is false, there is a bug, and that is something that needs to be found during the testing and debugging phase of programming.

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 subpackages, 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" commonly 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 is 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 could 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 some notations, such as `@Override`, 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.

Section 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, as the size of the input is increased." 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 $O(n^2)$ or $O(n)$ or $O(\log(n))$. 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 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 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 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 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 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 O(n).

Note that to say that an algorithm has run time O(f(n)) is to say that its run time is no bigger than some constant times f(n) (for large values of n). 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 O(n), it would also be correct to say that the run time is $O(n^2)$ or even $O(n^{10})$. 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 Ω(f(n)), read "Omega of f of n" or "Big 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 Ω(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).) O(f(n)) tells you something about the maximum amount of time that you might have to wait for an algorithm to finish; Ω(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 Ω(n) as well as O(n). When an algorithm has a run time that is both Ω(f(n)) and O(f(n)), its run time is said to be Θ(f(n)), read "Theta of f of n" or "Big Theta of f of n." (Theta is another letter from the Greek alphabet.) To say that the run time of an algorithm is Θ(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 void 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 $O(n^2)$, that is, the run time is no more than some constant times n^2. Since it is both $\Omega(n^2)$ and $O(n^2)$, the run time of the simpleBubbleSort algorithm is $\Theta(n^2)$.

You should be aware that some people use the notation $O(f(n))$ as if it meant $\Theta(f(n))$. That is, when they say that the run time of an algorithm is $O(f(n))$, 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, $O(f(n))$ 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 $O(f(n))$ 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.

It is also possible to talk about `best case` run time analysis, which looks at the **shortest** possible run time for all inputs of a given size. However, a best case analysis is only occasionally useful.

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 $O(f(n))$ 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 growth 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 7.5.1. 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 Θ(log(n)). (The average case run time is also Θ(log(n)).) By comparison, the linear search algorithm, which was also presented in Subsection 7.5.1 has a run time that is Θ(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.

Programming Exercises for Chapter 8

Exercise 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.

See the Solution

Exercise 8.2:

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 or with the `nextLine()` function of a *Scanner*. 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! The program should not end when that happens; it should just output an error message. 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.

See the Solution

Exercise 8.3:

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.

See the Solution

Exercise 8.4:

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.

See the Solution

Exercise 8.5:

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 clicks an "Enter" button or presses return, 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. (Recall: If you make a button into the default button for the window, then pressing return will be equivalent to clicking the button (see the end of Subsection 6.4.2).)

The program will need a *Canvas* for displaying the graph. To keep things simple, the canvas 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 x and y values in the range from -5 to 5 to the pixel coordinates that you need for drawing on the canvas. The formulas for the conversion are:

```
double a  =  ( (x + 5)/10 * width );
double b  =  ( (5 - y)/10 * height );
```

where `a` and `b` are the horizontal and vertical pixel coordinates on the canvas. The values of `width` and `height` give the size of the canvas.

See the Solution

Quiz on Chapter 8

Question 1:

Why do programming languages require that variables be declared before they are used? What does this have to do with correctness and robustness?

Question 2:

What is a *precondition*? Give an example.

Question 3:

Explain how preconditions can be used as an aid in writing correct programs.

Question 4:

Find a useful loop invariant for the `while` loop in the binary search algorithm (Subsection 7.5.1).

Question 5:

Java has a predefined class called *Throwable*. What does this class represent? Why does it exist?

Question 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*.

Question 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?

Question 8:

Some classes of exceptions are *checked exceptions* that require *mandatory exception handling*. Explain what this means.

Question 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.

Question 10:

Why should a subroutine throw an exception when it encounters an error? Why not just terminate the program?

Question 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?

Question 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 $O(f(n))$ 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];
}
```

See the Answers

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.

Contents of Chapter 9:

- Section 1: Recursion
- Section 2: Linked Data Structures
- Section 3: Stacks, Queues, and ADTs
- Section 4: Binary Trees
- Section 5: A Simple Recursive Descent Parser
- Programming Exercises
- Quiz on This Chapter

Section 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** (or **recursive method**) 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 7.5.1. 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 8.2.2, 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 illustrations are from a sample program from Chapter 12, TowersOfHanoiGUI.java, which displays an animation of the step-by-step solution of the problem; however, that program uses some techniques that you haven't learned yet.)

The stacks after a number of moves.

The problem is to move ten disks from Stack 0 to Stack 1, subject to the rules given above. 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.
 */
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.printf("Move disk 1 from stack %d to
stack %d%n",
                                from, 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.printf("Move disk %d from stack %d to
stack %d%n",
                               disks, from, 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, the subroutine should be called with the command `TowersOfHanoi(N,0,1,2)`. The subroutine is used in 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 disk 1 from stack 0 to stack 2
Move disk 2 from stack 0 to stack 1
Move disk 1 from stack 2 to stack 1
Move disk 3 from stack 0 to stack 2
Move disk 1 from stack 1 to stack 0
Move disk 2 from stack 1 to stack 2
```

```
Move disk 1 from stack 0 to stack 2
Move disk 4 from stack 0 to stack 1
Move disk 1 from stack 2 to stack 1
Move disk 2 from stack 2 to stack 0
Move disk 1 from stack 1 to stack 0
Move disk 3 from stack 2 to stack 1
Move disk 1 from stack 0 to stack 2
Move disk 2 from stack 0 to stack 1
Move disk 1 from stack 2 to stack 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. (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 program, mentioned above, there are two more demo programs that you might want to look at. Each program provides a visual demonstration of a recursive algorithm. In Maze.java, recursion is used to solve a maze. In LittlePentominos.java, it is used to solve a well-known kind of puzzle. (LittlePentominos.java also requires the file MosaicCanvas.java.) It would be useful to run the programs

and watch them for a while, but the source code uses some techniques that won't be covered until Chapter 12.

The Maze program first creates 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. When it can't find a solution from a square, it "backs up" out of that square and tries somewhere else. This common technique is referred to as **recursive backtracking**.

The LittlePentominos program 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—another example of recursive backtracking. A generalized version of the pentominos program with many more features can be found at https://math.hws.edu/eck/js/pentominos/pentominos.html.

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 7.5, 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.

To apply QuicksortStep to a list of numbers, select one of the numbers, 23 in this case. Arrange the numbers so that numbers less than 23 lie to its left and numbers greater than 23 lie to its right.

23 10 7 45 16 86 56 2 31 18

18 10 7 2 16 23 86 56 31 45

To finish sorting the list, sort the numbers to the left of 23, and sort the numbers to the right of 23. The nuber 23 itself is already in its final position and doesn't have to be moved again

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) {

       // Loop invariant (See Subsection
Subsection 8.2.3):  A[i] <= pivot
       // for i < lo, and A[i] >= pivot for i > hi.

       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 $\Theta(n*\log(n))$. 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. (One of these "rare" cases is when the

original array is already sorted or almost sorted, which is really not all that rare in practice. Applying the Quicksort algorithm as given above to a large sorted array will take a long time. One way to avoid that—with high probablility—is to pick the pivot for QuickSort step at random, rather than always using the first item.)

There are sorting algorithms that have both an average case and a worst case run time of Θ(n*log(n)). One example that is fairly easy to understand is MergeSort, which you can look up if you are interested.

9.1.4 Blob Counting

Next, we will look at counting the number of squares in a group of connected squares. I call a group of connected squares a "blob," and the sample program that we will consider is Blobs.java. The program displays a grid of small white, gray, and red squares. Here is a screenshot from the program, showing the grid of squares along with some controls:

The gray or red 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. In the picture, one of the blobs is shown in 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.

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(r,c)` to count the number of squares in a blob. The parameters `r` and `c` tell which blob to count, namely the blob that includes the square in a 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, if there are at least two squares in the blob, then 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 red 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 method that draws the grid of squares shows visited squares in red, which makes the blob visible.

The `getBlobSize()` method is also used for the other task that can be done by the program: counting all the 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++;
      }

   draw();  // Redraw the entire grid of squares.
                // 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()
```

Section 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 illustration. 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 *Colors* 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 that I wrote named *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 a sample program named ListDemo.java, so you can take a look at the code in context if you want.

One of the methods in the *StringList* class searches the list, looking for a specified string. If the string that we are looking for is `searchItem`, then 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. 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 2.3.3.)

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 into 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()
```

Section 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 be 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. Abstraction is an important general concept in computer science. We have seen other examples: control abstraction in Subsection 3.1.4 and procedural abstraction in Section 4.1. Here, we are considering **data abstraction**.

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 "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, but that is part of the private implementation of the ADT.)

```
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.

(As an aside, the nested *Node* class in *StackOfInts* could have been record class (Section 7.4), since nodes don't need to be modified after they are created. The implementation of `push()` would have to be changed to use the record class constructor. See StackOfInt.java, which uses this approach.)

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 7.2.4. 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:

```
import java.util.Arrays;  // For the Arrays.copyOf()
method.

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.
           items = Arrays.copyOf( items, 2*items.length );
       }
       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—including the fact that an attempt to pop from an empty stack will result in an *IllegalStateException*.

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** in this book. (These names are not completely standardized, in the way that "push" and "pop" are. For example, the operations are sometimes called "put" and "take.") 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 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 use another instance variable to store a pointer to the last node. 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
                              // when the queue is not
empty.

   /**
    * 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 we have just removed the
last item,
                         // then head is null.
      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
```

To help you follow what is being done here with the `tail` pointer, it might help to think in terms of a class invariant (Subsection 8.2.3): "If the queue is non-empty, then `tail` points to the last node in the queue." This invariant must be true at the beginning and at the end of each method call. For example, applying this to the `enqueue()` method, in the case of a non-empty list, the invariant tells us that a new node can be added to the back of the list simply by saying "`tail.next = newNode`. It also tells us how the value of `tail` must be set before returning from the method: it must be set to point to the node that was just added to the queue.

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, it is 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 try out the program. The program shows a grid of squares. Initially, all the squares are white. When you click on a white square, that square is "marked" by turning it red. The program than starts marking squares that are connected, horizontally or vertically, to squares that have already been marked. This process will eventually process every square in the grid. To understand how the program works, 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. You are done when there are no more index cards waiting in the pile to be processed.

In the program, 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, the program will never consider it again, since all of its neighbors have already been accounted for. Eventually, all the squares have been processed, all the squares are gray, 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. Here are three

pictures from the program, using the three different processing methods. In each case, the process was started by selecting a square near the middle of the grid. A stack is used for the picture on the left, a queue for the picture in the middle, and random selection for the picture on the right:

The patterns that are produced are very different. When using a stack, the program explores out as far as possible before it starts backtracking to look at previously encountered squares. With a queue, squares are processed roughly in the order of their distance from the starting point. When random selection is used, the result is an irregular blob, but it is a connected blob since a square can only be encountered if it is next to a previously encountered square.

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 list of encountered squares. 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. Again, the source code for the program is DepthBreadth.java.

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, parameters, and return address (the the point in the program where the computer should return to when the subroutine ends). 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.

In the case of a recursive subroutine, which calls itself, there can be several activation records on the stack for the same subroutine. This is how the computer keeps track of many recursive calls at the same time: It has a different activation record for each call.

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.

   System.out.println();

   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);
         System.out.println("   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.
            System.out.println("\nIllegal operator found
in input: " + op);
            return;
         }
         if (stack.isEmpty()) {
            System.out.println("   Stack is empty while
trying to evaluate " + op);
            System.out.println("\nNot enough numbers in
expression!");
```

```
            return;
         }
         y = stack.pop();
         if (stack.isEmpty()) {
            System.out.println("   Stack is empty while
trying to evaluate " + op);
            System.out.println("\nNot enough numbers in
expression!");
            return;
         }
         x = stack.pop();
         switch (op) {
                 case '+' -> answer = x + y;
                 case '-' -> answer = x - y;
                 case '*' -> answer = x * y;
                 case '/' -> answer = x / y;
                 default  -> answer = Math.pow(x,y);  //
(op must be '^'.)
         }
         stack.push(answer);
         System.out.println("   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.
      System.out.println("No expression provided.");
      return;
   }

   double value = stack.pop();  // Value of the
expression.
   System.out.println("   Popped " + value + " at end of
expression.");

   if (stack.isEmpty() == false) {
      System.out.println("   Stack is not empty.");
      System.out.println("\nNot enough operators for all
```

```
the numbers!");
      return;
   }

   System.out.println("\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.

Section 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 would 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 a binary tree, a child is either a "left child" or a "right child," and a node can have a right child even if it has no left 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 ) {  // Base case:  empty tree.
      return 0;  // An empty tree contains no nodes.
   }
   else {  // Recursive case:  root node plus two
subtrees.
      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. Subroutines that use postorder and inorder traversal to print the contents of a tree 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 items in the left subtree are 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 items in the left subtree are 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 in a way that makes both searching and insertion efficient. A binary tree used in this way is called a **binary sort tree** or **BST**.

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 or equal to 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 descendants. "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 Θ(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 Θ(n), which is much worse than Θ(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 Θ(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 *SortTreeDemo*, nodes in the binary tree are represented using the following static nested class, which includes a simple constructor to make 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.
```

```
            // Note that left and right pointers are null.
        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(runner.item) ) {
            // We've found the item.
         return true;
      }
      else if ( item.compareTo(runner.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 ugly 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, perhaps named *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 real 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.

Section 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 compilers.

A variety of different notations are used for BNF. The one that I will use here is fairly common. Although other notations are used, they express the same concepts.

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.

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. (For a discussion of robust handling of *TextIO* input, see Subsection 8.2.4.)

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. The rule 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();  // Read and evaluate the first
term.
   if (negative)
      val = -val;  // Apply the leading minus sign to the
first term.
   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();    // Read and
evaluate the next term.
      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 subclass of *ExpNote*, *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;
           // (for assert statements, see
Subsection 8.4.1)
      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! Should not
be possible.
      }
   }
   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();
      System.out.println("  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.
           // (The return value is a pointer to the root
of the tree.)
       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** or **abstract syntax 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.

Programming Exercises for Chapter 9

Exercise 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. Consider using the *BigInteger* class (see Exercise 8.2)

(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.)

See the Solution

Exercise 9.2:

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.

See the Solution

Exercise 9.3:

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.

See the Solution

Exercise 9.4:

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.

See the Solution

Exercise 9.5:

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.

See the Solution

Exercise 9.6:

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.

See the Solution

Exercise 9.7:

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**.)

See the Solution

Quiz on Chapter 9

Question 1:

Explain what is meant by a *recursive* subroutine.

Question 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.print("]");
    }
}
```

Show the output that would be produced by the subroutine calls `printStuff(0)`, `printStuff(1)`, `printStuff(2)`, and `printStuff(3)`.

Question 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.

Question 4:

Let *ListNode* be defined as in the previous problem. Suppose that `head` is a variable of type *ListNode* that points to the first node in a linked list. Write a code segment that will add the number 42 in a new node at the **end** of the list. Assume that the list is not empty. (There is no "tail pointer" for the list.)

Question 5:

List nodes can be used to build linked data structures that do not have the form of linked lists. Consider the list node class shown on the left and the code shown on the right:

```
class ListNode {
    int item;
    ListNode next;
    Listnode(int i) {
        item = i;
        next = null;
    }
}
```

```
ListNode one = new ListNode(10);
ListNode two = new ListNode(20);
ListNode three = new ListNode(30);
ListNode four = new ListNode(40);

one.next = two;
two.next = three;
three.next = four;
four.next = two;
```

Draw the data structure that is constructed by the code. What happens if you try to print the items in the data structure using the usual code for traversing a linked list:

```
ListNode runner = one;
while (runner != null) {
    System.out.println(runner.item);
    runner = runner.next();
}
```

Question 6:

What are the three operations on a *stack?*

Question 7:

What is the basic difference between a stack and a queue?

Question 8:

What is an *activation record*? What role does a stack of activation records play in a computer?

Question 9:

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.

Question 10:

Let *TreeNode* be the same class as in the previous problem. Write a recursive subroutine that makes a copy of a binary tree. The subroutine has a parameter that points to the root of the tree that is to be copied. The return type is *TreeNode*, and the return value should be a pointer to the root of the copy. The copy should consist of newly created nodes, and it should have exactly the same structure as the original tree.

Question 11:

What is a *postorder traversal* of a binary tree?

Question 12:

Suppose that a binary sort tree of integers is initially empty and that the following integers are inserted into the tree in the order shown:

```
5   7   1   3   4   2   6
```

Draw the binary sort tree that results. Then list the integers in the order that is produced by a post-order traversal of the tree.

Question 13:

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.)

Question 14:

Explain what is meant by *parsing* a computer program.

See the Answers

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.

You have already seen part of the solution in Section 7.3. That section introduced parameterized types and the *ArrayList* class. Parameterized types make it possible for the same class to work with many different kinds of data. This idea—that the same code can be used for a variety of data types—is called **generic programming**.

Contents of Chapter 10:

- Section 1: Generic Programming
- Section 2: Lists and Sets
- Section 3: Maps
- Section 4: Programming with the Java Collection Framework
- Section 5: Writing Generic Classes and Methods
- Section 6: Introduction the Stream API
- Programming Exercises
- Quiz on This Chapter

Section 10.1

Generic Programming

GENERIC PROGRAMMING refers to writing code that will work for many types of data. We encountered the alternative to generic programming in Subsection 7.2.4, 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*, *Color*, 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. Java's approach to this problem is parameterized types. As we saw in Section 7.3, the parameterized class *ArrayList* implements dynamic arrays. Since it is parameterized, there are types such as `ArrayList<String>` to represent dynamic arrays of *String*, `ArrayList<Color>` for dynamic arrays of colors, and more generally `ArrayList<T>` for any object type *T*. *ArrayList* is just one class, but the source code works for many different types. This is generic programming.

The *ArrayList* class is just one of many standard classes that are used for generic programming in Java. We will spend the next three sections looking at some of these classes and how they are used, and we'll see that there are also generic methods and generic interfaces. The classes and interfaces discussed in these sections are defined in the package `java.util`, and you will need import statements at the beginning of your programs to get access to them.

In Section 10.5, we will see that it is possible to define new generic classes, interfaces, and methods. Until then, we will stick to using Java's predefined generics. And in Section 10.6, we will look at **streams**, a relatively new feature of Java that makes extensive use of generics.

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 some other approaches to generic programming.

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 never 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. 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++. 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 <= i ; j++)
         if ( A[j] > A[position_of_max] )
```

```
            position_of_max = j;
        ItemType temp = A[i];
        A[i] = 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. (By the way, in C++, it is possible to write definitions of operators like > for any type, so that it is possible that > might work for values of type *Card*.)

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. Modern 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. Early versions of Java did not have parameterized types, but they did have classes to represent common data structures. Those classes were designed to work with *Objects*; that is, they could hold objects of any type, and there was no way to restrict the types of objects that could be stored in a given data structure. For example, *ArrayList* was not originally a parameterized type, so that any ArrayList could hold any type of object. This means that if `list` was an *ArrayList*, then `list.get(i)` would return a value of type *Object*. If the programmer was actually using the list to store *Strings*, the value returned by `list.get(i)` would have to be type-cast to treat it as a string:

```
String item = (String)list.get(i);
```

This is still a kind of generic programming, since one class can work for any kind of object, but it was closer in spirit to Smalltalk than it was to C++, since there is no way to do type checks at compile time. 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, which made it possible to create generic data structures that can be type-checked at compile time rather than at run time. For example, if `list` is of type `ArrayList<String>`, then the compiler will only allow objects of type *String* to be added to `list`. Furthermore, the return type of `list.get(i)` is *String*, so type-casting is not necessary. 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 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*. In that case, any type of object can be stored in the data structure. (But if that

is what you really want to do, it would be preferable to use the type `ArrayList<Object>`.)

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. Similarly, you can't type-cast to the type `ArrayList<String>`. 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. (However, although you can't have an array of `ArrayList<String>`, you **can** have an ArrayList of `ArrayList<String>`—with the type written as `ArrayList<ArrayList<String>>`—which is just as good or better.)

Fortunately, most programmers don't have to deal with such problems, since they turn up only in fairly advanced programming. Most people who use parameterized types will not encounter the problems, and they will get the benefits of type-safe generic programming with little difficulty.

It's worth noting that if the type parameter in a parameterized type can be deduced by the compiler, then the name of the type parameter can be omitted. For example, the word "String" is optional in the constructor in the following

statement, because the *ArrayList* that is created must be an `ArrayList<String>` to match the type of the variable:

```
ArrayList<String> words = new ArrayList<>();
```

10.1.4 The Java Collection Framework

As I've said, Java comes with a number of parameterized types that implement common data structures. This collection of data structure classes and interfaces is referred to as the **Java Collection Framework**, or **JCF**. We will spend the next few sections learning about the JCF.

The generic data structures in the Java Collection Framework 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 7.5.2. 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 an example of a parameterized type that has 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. For collections that are "sets," the defining property 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 an ArrayList. 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. (Remember that if *T* is a class, this includes objects belonging to a subclass of *T*, and if *T* is an interface, it includes any object that implements *T*.) The `add()` 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*. The test for equality is the same test that is used by `contains()`.
- `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<Color>* (as long as that collection supports removal of values):

```
Iterator<Color> iter = coll.iterator():
while ( iter.hasNext() ) {
    Color 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 *Color* 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<Color>* is used to iterate through a collection of *Color*; 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 7.1.1 for use with arrays and in Subsection 7.3.3 for use with ArrayLists. But in fact, a for-each loop can 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.

We have seen that 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 many 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`, the value of `str.equals(obj)` is `true` when `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.

   private int suit;  // Number from 0 to 3 that codes for
the suit --
                      // spades, diamonds, clubs or
hearts.
   private 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()` from 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. But in general, classes that implement *Comparable* should try to define `.equals()` and `compareTo()` so that `obj1.equals(obj2)` and `obj1.compareTo(obj2) == 0` always have the same value. (Some classes in the JCF use `compareTo()` rather than `equals()` to test objects for equality.)

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. 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 ) {
      int compareLast =
lastName.compareTo(other.lastName);
      if ( compareLast < 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 ( compareLast > 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.

(We have previously encountered *FullName* as a record class, in Subsection 7.4.1. Remember that record classes can implement interfaces, so we could define a *FullName* record class to implement *Comparable<FullName>*. In the record class, the `equals()` method would already be defined appropriately.)

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. Since *Comparator* is a functional interface, comparators are often defined by lambda expressions (see Section 4.5).

Note that it can often make sense to use a *Comparator* for which `obj1.equals(obj2)` does not always have the same value as `compare(obj1,obj2) == 0`. For example, when sorting addresses by zip code, you would use a *Comparator* that looks at the zip code field in the addresses that it compares.

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 in Section 7.3 about *ArrayLists*, Java's generic programming does not apply to the primitive types. This is because generic data structures can only hold objects, and values of primitive type are not objects. However, the "wrapper classes" that were introduced in Subsection 7.3.2 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 7.3.3.) 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, `Integer.valueOf(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>*.)

Section 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 main 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. 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. I will also briefly discuss a third category of collection known as a "priority queue."

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 uses linear search and 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—such as `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. The functions `linkedlist.remove()` and `linkedlist.pop()` are also defined, with the same meaning as `removeFirst()`.
- `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. The function `linkedlist.push(obj)` has the same meaning.
- `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.

There is some redundancy here, apparently 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 stack by using the methods named `push()` and `pop()`, or as a queue by using `add()` and `remove()` to implement the enqueue and dequeue operations.

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()`, `add(obj)`, and `set(obj)` that make it possible to move backwards in the list, to add an item at the current position of the iterator, and to replace one of the items in the list. 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()`. The method `iter.set(obj)` works similarly; it replaces the item that would be removed by `iter.remove()`. 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 a "tail pointer" 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 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, and it works when the new item has to be inserted at the beginning or at the end of the list. 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. However, 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. (You have already seen similar methods for arrays in the *Arrays* class.) 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.

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. That is, 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 `compare()` 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()` or `compare()` method. This can be a problem. Recall from Subsection 10.1.6 that the `compare()` method of a *Comparator* might consider two objects to be the same for the purpose of the comparison even though the objects are not equal. (This is less likely, but still possible for a properly defined `compareTo`() method.) For a *TreeSet* that uses the *Comparator*, this means that only **one** of those objects can be in the set. For example, if the *TreeSet* contains mailing addresses and if the `compare()` 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 the `compare()` or `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 the left and right subtrees of any node in the tree are at about the same size. 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  // words are output in sorted order
```

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<>( new TreeSet<>
(coll) );
```

This makes a sorted list of the elements of `coll` with no duplicates. This is a nice example of the power of generic programming. (Note that the type parameter, `String`, is optional in the two constructors in this example, since it can be deduced by the compiler.)

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 even possible for the order to change 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 Priority Queues

A **priority queue** is an abstract data type that represents a collection of items, where each item has an assigned "priority" that allows any two items to be compared. Operations on a priority queue include *add*, which adds an item to the collection, and *remove*, which removes and returns an item from the collection that has the minimum priority among items currently in the collection. (Maximum priority would also be possible, but in Java's version, the *remove* operation removes a minimum priority item.)

A simple implementation of priority queue could be achieved by using a linked list to store the items in the queue in order of increasing priority. In that case, *remove* would simply remove and return the first item in the list. However, *add* would have to insert the new item into its correct position in the list, an operation with average run time $\Theta(n)$, where n is the number of items in the list. In fact, priority queues can be implemented so that both *add* and *remove* have run time $\Theta(\log(n))$, which is much more efficient. (The efficient implementation uses something called a "heap," which is not to be confused with the heap where objects are created. I will not discuss the implementation here.)

The parameterized class *PriorityQueue<T>* implements a priority queue of objects of type *T*. This class implements the interface *Collection<T>*. So, if `pq` is a *PriorityQueue*, then it has all the methods defined in that interface. But the essential priority queue operations are

- `pq.add(obj)` — adds `obj` to the priority queue, where `obj` must be an object of type *T*
- `pq.remove()` — removes and returns an item of minimal priority. The return value is an object of type *T*. Throws an exception if the queue is empty.
- `pq.isEmpty()` — tests whether the priority queue is empty.

You've probably noticed that I haven't yet mentioned how the priority of items in the priority queue are determined. The situation is much like sorting: We need to be able to compare any two items in the queue. As with sorting, there are two solutions. If the items implement the *Comparable* interface,

then they can be compared using the `compareTo()` method from that interface. Alternatively, a *Comparator* object can be provided as a parameter to the *PriorityQueue* constructor. In that case, the Comparator's `compare()` method will be used to compare items.

Classes such as *String*, *Integer*, and *Date* that implement *Comparable* can be used in a priority queue. For example, a *PriorityQueue<String>* can be used to sort strings into lexicographic order: Just *add* all the strings to the priority queue, then *remove* them one-by-one. Since items are removed from the queue in order of priority, they will be removed in lexicographic order. Earlier, I showed how to use a *TreeSet* to sort and remove duplicates from a collection. A *PriorityQueue* can be used in a similar way to sort a collection without removing duplicates. For example, if `coll` is of type *Collection<String>*, then the following code segment will print all the items from `coll` in order, including duplicates:

```
PriorityQueue<String> pq = new PriorityQueue<>();
pq.addAll( coll );
while ( ! pq.isEmpty() ) {
    System.out.println( pq.remove() );
}
```

Note, by the way, that we can't use an iterator or a for-each loop to print the items in this example, since iterators and for-each loops do not traverse a priority queue in ascending order.

The sample program WordListWithPriorityQueue.java makes a sorted list of words from a file without removing duplicates, using a priority queue to hold the words. It is a minor modification of WordListWithTreeSet.java.

Although priority queues can be used for sorting, they have other natural applications. For example, consider the problem of scheduling "jobs" to be executed on a computer, where each job is assigned a priority and jobs with lower priority should always be executed before jobs with higher priority. Jobs can be placed into a priority queue as they are created. When the computer removes jobs from the queue for execution, they will be removed in order of increasing priority.

Section 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 an 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,Color>* would associate values of type *Color* 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`. 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.
- `map.containsKey(key)` — returns a boolean value that is `true` if the map associates some value to the specified `key`.
- `map.containsValue(value)` — returns a boolean value that is `true` if the map associates the specified `value` to some key.
- `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()` (or `compare()`) method is used to decide whether two keys are to be considered the same. This can have undesirable consequences if the comparison 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 7.5.2, 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 could be defined as

```
Map<String,String> directory = new TreeMap<>();
```

(using *TreeMap* so that the entries are kept in sorted order by name). 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 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 to avoid some of the ugly type names:

```
System.out.println("The map contains the following
associations:");
for ( Map.Entry<String,Double> entry : map.entrySet() ) {
   System.out.println( "   (" + entry.getKey() + "," +
entry.getValue() + ")" );
}
```

This is certainly a place where it would be convenient to use `var` to declare the variables (See Subsection 4.8.2). With `var`, the example using an iterator

becomes:

```
var entries = map.entrySet();
var entryIter = entries.iterator();
System.out.println("The map contains the following
associations:");
while (entryIter.hasNext()) { . . .
```

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 *K*, 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 `phoneBook` 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 `phoneBook` where the name begins with "M" as follows:

```
Map<String,String> ems = phoneBook.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 ( var entry : ems.entrySet() ) {
         // Note: type for entry is
Map.Entry<String,String>
         // but it's easier to user var to declare the
variable!
      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. For example, suppose that a database of scheduled events is stored in a map of type *TreeMap<DateTime,Event>* in which the a key gives the date and time of an event, and suppose you want a listing of all events that are scheduled for some time on July 4, 2022. Just make a submap containing all keys in the range from 12:00 AM, July 4, 2022

to 12:00 AM, July 5, 2022, 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, there are no values, 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 through 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 if the hash table is working properly, 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 are two items with hash code 0, no items with hash code 1, one item 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 must 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 a 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.

Section 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<>();
```

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.

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* representing page numbers. 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, but if you think about the data structure that we want to represent, it makes sense.

To make an index, we need to start with an empty *TreeMap*. We then work through the book, and we insert into the map every reference that we want to be in the index. 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<>();  // Create the map object.
```

(Note that even for this complex type, the type parameters can be omitted from the constructor.)

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<>();
       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 );
   }
}
```

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 10.6 presents a problem that is almost identical to the indexing problem.

(By the way, the `printIndex()` method could have used `var` to declare all of its local variables. This would have avoided the complex type names, but you would still need to be aware of the types, since you have to know what methods, such as `entry.getKey()`, can be used with the variables.)

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.
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 negative, zero, or positive, depending on whether `obj1` is less than, equal to, or greater than `obj2`. We want to compare two *Strings* ignoring case. The *String* class already includes a method that does that, `compareToIgnoreCase()`. But to use a non-default comparison in a *TreeMap*, we need an object of type *Comparator<String>*. Since *Comparator* is a functional interface, an easy way to specify the comparator is to use a lambda expression:

```
(a,b) -> a.compareToIgnoreCase(b)
```

To solve our indexing problem, we just need to pass this comparator as a parameter to the *TreeMap* constructor:

```
index = new TreeMap<>( (a,b) -> a.compareToIgnoreCase(b)
);
```

In fact, since the lambda expression is just calling a method that already exists in the *String* class, it could actually be given as a method reference (see Subsection 4.5.4):

```
index = new TreeMap<>( String::compareToIgnoreCase );
```

This does work. However, there 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 a very 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<>();
```

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, we can copy the *WordData* objects into a list and then use the generic method `Collections.sort(list,comparator)`, which specifies a comparator as its second parameter. Since we want to sort the data into **decreasing** order by count, we want a comparator for two *WordData* values `a` and `b` that puts `a` before `b` if `a.count > b.count`. You should check that the following lambda expression defines a comparator that will work here:

```
(a,b) -> b.count - a.count
```

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<>(
words.values() );
Collections.sort( wordsByFrequency, (a,b) -> b.count -
a.count );
```

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 2.4.4.

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!

Section 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<>();
   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 *Colors* 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<>();
   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<Color>*. 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<>
();
   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<K,V>*. 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:

```
public 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<>("Red",
Color.RED);
Pair<Double,Double> coordinates = new Pair<>(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.

Record classes can also be generic. For example, a generic record class version of *Pair* could be defined simply as

```
public record Pair(T a, S b) { }
```

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 non-primitive 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*. 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 "`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 *Buttons*, ...

10.5.3 Wildcard Types

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 concat(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 formal parameter, `shapes`, is declared to be of type *Collection<? 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.)

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 subclass 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 is equal to or 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`". This ensures 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 in 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 have seen that *Runnable* is an `interface` that defines the method `public void run()`. 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 *Collection<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 formal 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 *ControlGroup*. For example, the parameterized type *ControlGroup<Button>* would represent a group of *Buttons*, while *ControlGroup<Slider>* would represent a group of *Sliders*. 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 disableAll() {
    .
    .  // Call c.setDisable(true) for every control, c, in
the group.
    .
}
```

The problem is that the `setDisable()` method is defined in a *Control* object, but not for objects of arbitrary type. It wouldn't make sense to allow types such as *ControlGroup<String>* or *ControlGroup<Integer>*, since *Strings* and *Integers* don't have `setDisable()` methods. We need some way to restrict the type parameter *T* in *ControlGroup<T>* so that only *Control*

and subclasses of *Control* are allowed as actual type parameters. We can do this by using the **bounded type** "`T extends Control`" instead of a plain "`T`" in the definition of the class:

```
public class ControlGroup<T extends Control> {
   private ArrayList<T> components; // For storing the
components in this group.
   public void disableAll( ) {
      for ( Control c : components ) {
         if (c != null)
            c.setDisable(true);
      }
   }
   public void enableAll( ) {
      for ( Control c : components ) {
         if (c != null)
            c.setDisable(false);
      }
   }
   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 Control`" on `T` makes it illegal to create the parameterized types *ControlGroup<String>* and *ControlGroup<Integer>*, since the actual type parameter that replaces "`T`" is required to be either *Control* itself or a subclass of *Control*. With this restriction, we know—and, more important, the compiler knows—that the objects in the group are of type *Control*, so that the operation `c.setDisable()` is 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*; 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 *ControlGroup* class, one could write a free-standing generic `static` method that can disable any collection of *Controls* as follows:

```
public static <T extends Control> void
disableAll(Collection<T> comps) {
   for ( Control c : comps )
      if (c != null)
         c.setDisable(true);
}
```

Using "`<T extends Control>`" as the formal type parameter means that the method can only be called for collections whose base type is `Control` or some subclass of `Control`, such as `Button` or `Slider`.

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 disableAll(Collection<? extends
Control> comps) {
   for ( Control c : comps )
      if (c != null)
         c.setDisable(true);
}
```

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 parameter 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 = sortedList.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, ...
```

Section 10.6

Introduction the Stream API

AMONG ITS NEW FEATURES, Java 8 introduced a **stream API**, which represents a new way of expressing operations on collections of data. A major motivation for the new approach was to make it possible for the Java compiler to "parallelize" a computation, that is, to break it into pieces that can be run simultaneously on several processors. Doing so can significantly speed up the computation. Chapter 12 will discuss parallel programming in Java using threads. Using threads directly can be difficult and error-prone. The stream API offers the possibility of parallelizing some kinds of computation automatically and safely, and it is not surprising that it has generated a lot of interest on those grounds.

The classes and interfaces that define the stream API are defined in package `java.util.stream`. *Stream* is an interface in that package that represents streams and defines the basic operations on streams.

A **stream** is simply a sequence of data values. A stream can be created from a *Collection*, from an array, or from a variety of other data sources. The stream API provides a set of operators that operate on streams. (The API is covered in this chapter because it makes extensive use of generic programming and parameterized types.) To perform some computation using the stream API means creating a stream to get the data from some source, and then applying a sequence of stream operations that will produce the result that you want. Once a stream has been used in this way, it cannot be reused. Of course, you can usually make another stream from the same data source if you want to use it in another computation.

Expressing a computation as a sequence of stream operations requires a new kind of thinking, and it takes some getting used to. Let's start with an example. Suppose that `stringList` is a large `ArrayList<String>`, where none of the elements are `null`, and you want to know the average length of the strings in the list. This can be done easily with a for-each loop:

```
int lengthSum = 0;
for ( String str : stringList ) {
    lengthSum = lengthSum + str.length();
}
double average = (double)lengthSum / stringList.size();
```

To do the same thing with the stream API, you could use:

```
int lengthSum = stringList.parallelStream()
                          .mapToInt( str -> str.length() )
                          .sum();
double average = (double)lengthSum / stringList.size();
```

In this version, `stringList.parallelStream()` creates a stream consisting of all the elements of the list. The fact that it is a "parallelStream" makes it possible to parallelize the computation. The method `mapToInt()` applies a **map** operation to the stream of strings. That is, it takes each string from the stream and applies a function to it; in this case, the function computes the length of the string, giving a value of type int. The result of the map operation is to produce a new stream, this time a stream of integers, consisting of all the outputs from the function. The final operation, `sum()`, adds up all the numbers in the stream of integers and returns the result.

The net result is that we've added up the lengths of all the strings in the list. Because of the potential for parallelization, the stream version might be substantially faster than the for loop version. In practice, there is significant overhead involved in setting up and manipulating streams, so the list would have to be fairly large before you would see any speedup. In fact, for small lists, the stream version will almost certainly take longer than the for loop.

The stream API is complex, and I can only give a basic introduction to it here —but hopefully enough to convey some of its spirit.

10.6.1 Generic Functional Interfaces

Many stream operators take parameters, which are often given as lambda expressions. The `mapToInt()` operator in the above example takes a parameter representing a function from *String* to int. The type for that

parameter is given by a parameterized functional interface, `ToIntFunction<T>`, from package `java.util.function`. This interface represents the general idea of a function that takes an input of type *T* and outputs an int. If you were to look at the definition of that interface, it would be something like

```
public interface ToIntFunction<T> {
    public int applyAsInt( T x );
}
```

`Stream<T>` is also a parameterized interface. In the above example, `stringList` is of type `ArrayList<String>`, and the stream that is created by `stringList.parallelStream()` is of type `Stream<String>`. When the `mapToInt()` operator is applied to that stream, it expects a parameter of type `ToIntFunction<String>`. The lambda expression "`str -> str.length()`" maps a *String* to an int, so it represents a value of the correct type. Fortunately, you don't need to think about all of that to use the stream API: All you need to know is that to convert a stream of strings to a stream of integers using `mapToInt`, you need to provide a function that maps strings to ints. However, if you want to read the API documentation, you will have to deal with parameter types similar to *ToIntFunction*.

The package `java.util.function` contains a large number of generic functional interfaces. Many of them, like *ToIntFunction*, are parameterized types, and they are all generic in that they represent very generic functions, with no set meaning. For example, the functional interface *DoubleUnaryOperator* represents the general idea of a function from double to double. This interface is essentially the same as my example *FunctionR2R* from Subsection 4.5.2 (except for the name of the function that it defines, which is often irrelevant).

The interfaces in `java.util.function` are used to specify parameter types for many stream operators as well as for other built-in functions in the Java API, and you can certainly use them to specify parameter types for your own subroutines as well. I will discuss some of them here. Most of the others are variations on the ones that I cover.

The general term **predicate** refers to a function whose return type is boolean. The functional interface `Predicate<T>` defines a boolean-valued function `test(t)` with a parameter of type *T*. This interface is used, for example, as the parameter type for the method `removeIf(p)`, which is defined for any *Collection*. For example, if `strList` is of type `LinkedList<String>`, then you can remove all `null` values from the list simply by saying

```
strList.removeIf( s -> (s == null) );
```

The parameter is a `Predicate<String>` that tests whether its input, `s`, is `null`. The `removeIf()` method removes all elements from the list for which the value of the predicate is `true`.

A predicate for testing int values could be represented by the type `Predicate<Integer>`, but that introduces the overhead of autoboxing every int in a wrapper of type *Integer*. To avoid that overhead, the package `java.util.function` has the functional interface *IntPredicate*, which defines the boolean-valued function `test(n)`, where `n` is of type int. Similarly, it defines *DoublePredicate* and *LongPredicate*. This is typical of how the stream API deals with primitive types. For example, it defines *IntStream* to represent a stream of ints as a more efficient alternative to `Stream<Integer>`.

The functional interface `Supplier<T>` defines a function, `get()` with no parameters and a return type of *T*. It represents a source of values of type *T*. There is a companion interface `Consumer<T>` that defines the `void` function `accept(t)` with a parameter of type *T*. There are also specialized versions for primitive types, including *IntSupplier*, *IntConsumer*, *DoubleSupplier* and *DoubleConsumer*. I will give examples of using suppliers and consumers below.

`Function<T,R>` represents functions from values of type *T* to values of type *R*. This functional interface defines the function `apply(t)`, where `t` is of type *T* and the return type is *R*. The interface `UnaryOperator<T>` is essentially `Function<T,T>`; that is, it represents a function whose input and output types are the same. Note that *DoubleUnaryOperator* is a specialized version of `UnaryOperator<Double>`, and of course there is also *IntUnaryOperator*.

Finally, I will mention `BinaryOperator<T>` and its specializations such as *IntBinaryOperator*. The interface `BinaryOperator<T>` defines the function `apply(t1,t2)` where `t1` and `t2` are both of type *T* and the return type is also *T*. Binary operators include things like addition of numbers or concatenation of strings.

10.6.2 Making Streams

To use the stream API, you have to start by creating a stream. There are many ways to make streams.

There are two basic types of streams, **sequential streams** and **parallel streams**. The difference is that operations on parallel streams can, potentially, be parallelized while the values in a sequential stream are always processed sequentially, in a single process, as they would be by a `for` loop. (It might not be clear why sequential streams should exist, but some operations cannot be safely parallelized.) It is possible to convert a stream from one type to the other type. If `stream` is a *Stream*, then `stream.parallel()` represents the same stream of values, but converted to a parallel stream (if it was not already parallel). Similarly, `stream.sequential()` is a sequential stream with the same values as `stream`.

We have already seen that if `c` is any *Collection*, then `c.parallelStream()` is a stream whose values are the values from the collection. As you might suspect, it is a parallel stream. The method `c.stream()` creates a sequential stream of the same values. This works for any collection, including lists and sets. You could also get the parallel stream by calling `c.stream().parallel()`.

An array does not have a `stream()` method, but you can create a stream from an array using a `static` method in class *Arrays* from package `java.util`. If `A` is an array, then

```
Arrays.stream(A)
```

is a sequential stream containing the values from the array. (To get a parallel stream, use `Arrays.stream(A).parallel()`.) This works for arrays of objects and for arrays of the primitive types int, double, and long. If `A` is of type `T[]`, where *T* is an object type, then the stream is of type `Stream<T>`. If `A` is an array of int, the result is an *IntStream*, and similarly for double and long.

Suppose `supplier` is of type `Supplier<T>`. It should be possible to create a stream of values of type *T* by calling `supplier.get()` over and over. That stream can in fact be created using

```
Stream.generate( supplier )
```

The stream is sequential and is effectively infinite. That is, it will continue to produce values forever or until trying to do so produces an error. Similarly, `IntStream.generate(s)` will create the stream of int values from an *IntSupplier*, and `DoubleStream.generate(s)` creates a stream of doubles from a *DoubleSupplier*. For example,

```
DoubleStream.generate( () -> Math.random() )
```

creates an infinite stream of random numbers. In fact, you can get a similar stream of random values from a variable, `rand`, of type *Random* (see Subsection 5.3.1): `rand.doubles()` is an infinite stream of random numbers in the range 0 to 1. If you only want a finite number of random numbers, use `rand.doubles(count)`. The *Random* class has other methods for creating streams of random doubles and ints. You will find other methods that create streams in various standard classes.

The *IntStream* interface defines a method for creating a stream containing a given range of integer values. The stream

```
IntStream.range( start, end )
```

is a sequential stream containing the values `start`, `start+1`, ..., `end-1`. Note that `end` is not included.

Some additional methods for making streams have been introduced in newer versions of Java. For example, in Java 11, for a *Scanner*, `input`, the method

`input.tokens()` creates a stream consisting of all the strings that would be returned by calling `input.next()` over and over. And for a *String*, `str`, that contains multiple lines of text, Java 11 added `str.lines()` that creates a stream consisting of the lines from the string.

10.6.3 Operations on Streams

Some operations on a stream produce another stream. They are referred to as "intermediate operations" because you will still have to do something with the resulting stream to produce a final result. "Terminal operations" on the other hand apply to a stream and produce a result that is not a stream. The general pattern for working with streams is to create a stream, possibly apply a sequence of intermediate operations to it, and then apply a terminal operation to produce the desired final result. In the example at the beginning of this section, `mapToInt()` is an intermediate operation that converted the stream of strings into a stream of ints, and `sum()` is a terminal operation that found the sum of the numbers in the stream of ints.

The two most basic intermediate operations are **filter** and **map**. A filter applies a *Predicate* to a stream, and it creates a new stream consisting of the values from the original stream for which the predicate is true. For example, if we had a boolean-valued function `isPrime(n)` that tests whether an integer `n` is prime, then

```
IntSteam.range(2,1000).filter( n -> isPrime(n) )
```

would create a stream containing all the prime numbers in the range 2 to 1000. (I'm not saying this is a *good* way to produce those numbers!)

A map applies a *Function* to each value in a stream, and creates a stream consisting of the output values. For example, suppose that `strList` is an `ArrayList<String>` and we would like a stream consisting of all the non-null strings in the list, converted to lower case:

```
strList.stream().filter( s -> (s != null) ).map( s ->
s.toLowerCase() )
```

The specializations `mapToInt()`, `mapToDouble()`, and `mapToLong()` exist to map *Streams* into *IntStreams*, *DoubleStreams*, and *LongStreams*.

Here are a few more intermediate operations on a stream, `S`, that can be useful: `S.limit(n)`, where `n` is an integer, creates a stream containing only the first `n` values from `S`. (If `S` has fewer than `n` values, then `S.limit(n)` is the same as `S`.) `S.distinct()` creates a stream from the values of `S` by omitting duplicate values, so that all the values in `S.distinct()` are different. And `S.sorted()` creates a stream containing the same values as `S`, but in sorted order; to sort items that do not have a natural ordering, you can provide a *Comparator* as a parameter to `sorted()`. (Comparators are discussed in Subsection 10.1.6.) Note that `S.limit(n)` can be especially useful for truncating what would otherwise be an infinite stream, such as a stream generated from a *Supplier*.

To actually get anything done with a stream, you need to apply a terminal operation at the end. The operator `forEach(c)` applies a *Consumer*, `c`, to each element of the stream. The result is not a stream, since consumers do not produce values. The effect of `S.forEach(c)` on a stream `S` is simply to do something with each value in the stream. For example, we have a whole new way to print all the strings in a list of strings:

```
stringList.stream().forEach( s -> System.out.println(s) );
```

For parallel streams, the consumer function is not guaranteed to be applied to the values from the stream in the same order that they occur in the stream. If you want that guarantee, you can use `forEachOrdered(c)` instead of `forEach(c)`.

If we want to print out only some of the strings, say those that have length at least 5, and if we want them in sorted order, we can apply some filters:

```
stringList.stream()
          .filter( s -> (s.length() >= 5) )
          .sorted()
          .forEachOrdered( s -> System.out.println(s) )
```

Some terminal operations output a single value. For example, `S.count()` returns the number of values in the stream `S`. And *IntStreams*, *LongStreams*, and *DoubleStreams* have the terminal operation `sum()`, to compute the sum of all the values in the stream. Suppose, for example, that you would like to test the random number generator by generating 10000 random numbers and counting how many of them are less than 0.5:

```
long half = DoubleStream.generate( Math::random )
                        .limit(10000)
                        .filter( x -> (x < 0.5) )
                        .count();
```

Note that `count()` returns a long rather than an int. Also note that I've used the method reference `Math::random` here instead of the equivalent lambda expression "`() -> Math.random()`" (see Subsection 4.5.4). If you are having trouble reading things like this, keep in mind that the pattern is: Create a stream, apply some intermediate operations, apply a terminal operation. Here, an infinite stream of random numbers is generated by calling `Math.random()` over and over. The operation `limit(10000)` truncates that stream to 10000 values, so that in fact only 10000 values are generated. The `filter()` operation only lets through numbers `x` such that `x < 0.5` is true. And finally, `count()` returns the number of items in the resulting stream.

A `Stream<T>` also has terminal operations `min(c)` and `max(c)` to return the smallest and largest values in the stream. The parameter, *c*, is of type `Comparator<T>`; it is used for comparing the values. However, the return type of `min()` and `max()` is a little peculiar: The return type is `Optional<T>`, which represents a value of type *T* that might or might not exist. The problem is that an empty stream does not have a largest or smallest value, so the minimum and maximum of an empty stream do not exist. An *Optional* has a `get()` method that returns the value of the *Optional*, if there is one; it will throw an exception if the *Optional* is empty. For example, if `words` is a `Collection<String>`, you can get the longest string in the collection with

```
String longest = words.parallelStream()
                      .max( (s1,s2) -> s1.length() -
s2.length() )
                      .get();
```

But this will throw an exception if the collection is empty. (The boolean-valued method `isPresent()` in an *Optional* can be used to test whether the value exists.)

Similarly, *IntStream*, *LongStream*, and *DoubleStream* provide terminal operations `min()` and `max()` that return values of type *OptionalInt*, *OptionalLong*, and *OptionalDouble*. Each of these classes also has an `average()` method that returns an *OptionalDouble*.

The terminal operators `allMatch(p)` and `anyMatch(p)` take a predicate as parameter and compute a boolean value. The value of `allMatch(p)` is true if the predicate, `p`, is true for every value in the stream to which it is applied. The value of `anyMatch(p)` is true if there is at least one value in the stream for which `p` is true. Note that `anyMatch()` will stop processing, and will return `true` as its output, if it finds a value that satisfies the predicate. And `allMatch()` will stop processing if it finds a value that does not match the predicate.

Many terminal operations that compute a single value can be expressed in terms of a more general operation, **reduce**. A reduce operation combines the values from a stream using a *BinaryOperator*. For example, a sum is computed by a reduce operation in which the binary operation is addition. The binary operator should be associative, which means that the order in which the operator is applied doesn't matter. There is no built-in terminal operator to compute the product of the values in a stream, but we can do that directly with `reduce`. Suppose, for example, that `A` is an array of double, and we want the product of all the non-zero elements in `A`:

```
double multiply = Arrays.stream(A).filter( x -> (x != 0) )
                                   .reduce( 1, (x,y) -> x*y
);
```

The binary operator here maps a pair of numbers `(x,y)` to their product `x*y`. The first parameter to `reduce()` is an "identity" for the binary operation. That is, it is a value such that `1*x = x` for any `x`. The maximum of a stream of double could be computed with `reduce()` by using `reduce(Double.NEGATIVE_INFINITY, Math::max)`.

The last major terminal operation is `collect(c)`, a very general operation which collects all of the values in the stream into a data structure or a single summary result of some type. The parameter, `c` is something called a collector. The collector will ordinarily be given by one of the `static` functions in the *Collectors* class. This can get very complicated, and I will only give a couple of examples. The function `Collectors.toList()` returns a *Collector* that can be used with `collect()` to put all of the values from the stream into a *List*. For example, suppose that `A` is an array of non-null *Strings*, and we want a list of all the strings in `A` that begin with the substring "Fred":

```
List<String> freds = Arrays.stream(A)
                            .filter( s ->
s.startsWith("Fred") )
                            .collect( Collectors.toList()
);
```

That's actually pretty easy! Even more useful are collectors that group the items from a stream according to some criterion. The collector `Collectors.groupingBy(f)` takes a parameter, `f`, whose type is specified by the functional interface *Function<T,S>*, representing a function from values of type *T* to values of type *S*. When used with `collect()`, `Collectors.groupingBy(f)` operates on a stream of type *Stream<T>*, and it separates the items in the stream into groups, based on the value of the function `f` when applied to the items. That is, all the items, `x`, in a given group have the same value for `f(x)`. The result is a *Map<S,List<T>>*. In this map, a key is one of the function values, `f(x)`, and the associated value for that key is a list containing all the items from the stream to which `f` assigns that function value.

An example will clarify things. Suppose we have an array of people, where each person has a first name and a last name. And suppose that we want to put the people into groups, where each group consists of all the people with a given last name. A person can be represented by an object of type *Person* that contains instance variables named *firstname* and *lastname*. Let's say that `population` is a variable of type `Person[]`. Then `Arrays.stream(population)` is a stream of *Persons*, and we can group the people in the stream by last name with the following code:

```
Map<String, List<Person>> families;
families = Arrays.stream(population)
                 .collect(Collectors.groupingBy( person ->
person.lastname ));
```

Here, the lambda expression, `person -> person.lastname`, defines the grouping function. The function takes a *Person* as input and outputs a *String* giving that person's last name. In the resulting *Map*, `families`, a key is one of the last names from the *Persons* in the array, and the value associated with that last name is a *List* containing all the *Persons* with that last name. We could print out the groups as follows:

```
for ( String lastName : families.keySet() ) {
    System.out.println("People with last name " + lastName
+ ":");
    for ( Person name : families.get(lastName) ) {
        System.out.println("    " + name.firstname + " " +
name.lastname);
    }
    System.out.println();
}
```

Although the explanation is a bit long-winded, the result should be reasonably easy to understand.

10.6.4 An Experiment

Most of the examples of using streams that I have given so far are not very practical. In most cases, a simple for loop would have been just as easy to write and probably more efficient. That's especially true since I've mostly used sequential streams, and most of the examples cannot be efficiently parallelized. (A notable exception is the reduce operation, which is important precisely because it parallelizes well.) Let's look at an example where the stream API is applied to a long computation that might get some real speedup with parallelization. The problem is to compute a Riemann sum. This is something from Calculus, but you don't need to understand anything at all about what it means. Here is a traditional method for computing the desired sum:

```
/**
 * Use a basic for loop to compute a Riemann sum.
 * @param f  The function that is to be summed.
 * @param a  The left endpoint of the interval over which
f is summed.
 * @param b  The right endpoint.
 * @param n  The number of subdivisions of the interval.
 * @return   The value computed for the Riemann sum.
 */
private static double riemannSumWithForLoop(
        DoubleUnaryOperator f, double a, double b, int n)
{
    double sum = 0;
    double dx = (b - a) / n;
    for (int i = 0; i < n; i++) {
        sum = sum + f.applyAsDouble(a + i*dx);
    }
    return sum * dx;
}
```

The type for the first parameter is a functional interface, so we could call this method, for example, with

```
reimannSumWithForLoop( x -> Math.sin(x), 0, Math.PI, 10000
)
```

How can we apply the stream API to this problem? To imitate the `for` loop, we can start by generating the integers from 0 to `n` as a stream, using `IntStream.range(0,n)`. This gives a sequential stream. To enable parallelism, we have to convert it to a parallel stream by applying the `.parallel()` operation. To compute the values that we want to sum up, we can apply a map operation that maps the stream of ints to a stream of doubles by mapping the integer `i` to `f.applyAsDouble(a+i*dx)`. Finally, we can apply `sum()` as the terminal operation. Here is a version of the Riemann sum method that uses a parallel stream:

```
private static double riemannSumWithParallelStream(
        DoubleUnaryOperator f, double a, double b, int n)
{
    double dx = (b - a) / n;
    double sum = IntStream.range(0,n)
                          .parallel()
                          .mapToDouble( i ->
f.applyAsDouble(a + i*dx) )
```

```
                                .sum();
    return sum * dx;
}
```

I also wrote a version `riemannSumWithSequentialStream()`, that leaves out the `.parallel()` operator. All three versions can be found in the sample program RiemannSumStreamExperiment.java. The main routine in that program calls each of the three methods, using various values for `n`. It times how long each method takes to compute the sum, and reports the result.

As might be expected, I found that the version that uses a sequential stream is uniformly slower than the other versions. The sequential stream version does essentially the same thing as the for loop version, but with the extra overhead involved with creating and manipulating streams. The situation for parallel streams is more interesting, and the results depend on the machine on which the program is executed. On one old machine with four processors, the for loop version was faster for `n` = 100,000, but the parallel version was much faster for 1,000,000 items or more. On another machine, the parallel version was faster for 10,000 or more items. Note that there is a limit to how much faster the parallel version can be. On a machine with K processors, the parallel version cannot be more than K times faster than the sequential version, and will probably in reality be somewhat slower than that. I encourage you to try out the sample program on your own computer!

It is even conceivable (or at least this is a goal of the stream API) that you have a machine on which Java can run parallel code on your graphics card, making use of the many processors that it contains. If that happens, you might see a very large speedup.

Programming Exercises for Chapter 10

Exercise 10.1:

Rewrite the *PhoneDirectory* class from Subsection 7.5.2 so that it uses a *TreeMap* to store directory entries, instead of an array. (Doing this was suggested in Subsection 10.3.1.) You should also write a short program to test the class.

See the Solution

Exercise 10.2:

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.) For input and output, a set of such integers will be written 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.print(A)`. I've chosen the syntax for sets to be the same as that used by the system for outputting a set.)

See the Solution

Exercise 10.3:

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 are **not** required to increase the size of the table when it becomes too full.

You should also write a short program to test your solution.

See the Solution

Exercise 10.4:

A **predicate** is a boolean-valued function with one parameter. Java has the parameterized functional interface `Predicate<T>`, from package `java.util.function`, to represent predicates. The definition of `Predicate<T>` could be:

```
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. Java already has some methods that use predicates, such as the `removeIf(p)` method that is defined for any *Collection*. (See Subsection 10.6.1). However, this exercise asks you to write a few similar methods yourself. 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. You should **not** use the stream API for this exercise.

```
public static <T> void remove(Collection<T> coll,
Predicate<T> pred)
   // Remove every object, obj, from coll for which
pred.test(obj)
   // is true.  (This does the same thing as
coll.removeIf(pred).)

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.
```

See the Solution

Exercise 10.5:

This is a short exercise in using the stream API. Suppose that the class *Score* is defined as

```
/**
 * Data for one student about a score on a test.
 */
private record ScoreInfo(
    String lastName,
    String firstName,
    int score
) { };
```

defined here as a record class for convenience (see Section 7.4). And suppose that `scoreData` is an array of *ScoreInfos* containing information about the scores of students on a test. Use the stream API to do each of the following tasks:

- print the number of students (without using `scoreData.length`)
- print the average score for all of the students
- print the number of students who got an A (score greater than or equal to 90)

- use the `collect()` stream operation to create a *List<String>* that contains the names of students whose score was less than 70; the names should be in the form first name followed by last name
- print the names from the *List* that was generated in the previous task
- print out the students' names and scores, ordered by last name
- print out the students' names and scores, ordered by score

You can put all of the code in `main()` routine and include *ScoreInfo* as a nested class. Do not use **any** for loops or other control structures. Do everything using the stream API. For testing your code, you can use this array:

```
private static ScoreInfo[] scoreData = new ScoreInfo[] {
        new ScoreInfo("Smith","John",70),
        new ScoreInfo("Doe","Mary",85),
        new ScoreInfo("Page","Alice",82),
        new ScoreInfo("Cooper","Jill",97),
        new ScoreInfo("Flintstone","Fred",66),
        new ScoreInfo("Rubble","Barney",80),
        new ScoreInfo("Smith","Judy",48),
        new ScoreInfo("Dean","James",90),
        new ScoreInfo("Russ","Joe",55),
        new ScoreInfo("Wolfe","Bill",73),
        new ScoreInfo("Dart","Mary",54),
        new ScoreInfo("Rogers","Chris",78),
        new ScoreInfo("Toole","Pat",51),
        new ScoreInfo("Khan","Omar",93),
        new ScoreInfo("Smith","Ann",95)
};
```

See the Solution

Exercise 10.6:

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.

See the Solution

Exercise 10.7:

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) {
          // (This uses the "switch expression" syntax)
      return switch(functionCode) {
              case SIN -> Math.sin(x);
              case COS -> Math.cos(x);
              case TAN -> Math.tan(x);
              case ABS -> Math.abs(x);
              case SQRT -> Math.sqrt(x);
```

```
                default -> 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(Function.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)`.

See the Solution

Quiz on Chapter 10

Question 1:

What is meant by *generic programming* and what is the alternative?

Question 2:

Why can't you make an object of type *LinkedList<int>*? What should you do instead?

Question 3:

What is an *iterator* and why are iterators necessary for generic programming?

Question 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.

Question 5:

Interfaces such as *List*, *Set*, and *Map* define *abstract data types*. Explain what this means.

Question 6:

What is the fundamental property that distinguishes *Sets* from other types of *Collections*?

Question 7:

What is the essential difference in functionality between a *TreeMap* and a *HashMap*?

Question 8:

Explain what is meant by a *hash code*.

Question 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;
   }
}
```

Also, rewrite the resulting *Date* class as a record class. (See Section 7.4.)

Question 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, 2021.

Question 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.

Question 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.

Question 13:

Suppose that `words` is an array of *Strings*. Explain what is done by the following code:

```
long n = Arrays.stream(words)
               .filter( w -> (w != null) )
               .map( w -> w.toLowerCase() )
               .distinct()
               .count();
```

Question 14:

Use the stream API to print all the even integers from 2 to 20. Start with `IntStream.range` and apply a `filter` operation.

Question 15:

Write a generic method `countIf(c,t)` with type parameter `<T>`, where the first parameter, `c`, is of type `Collection<T>`, and the second parameter, `p`, is of type `Predicate<T>`. The method should return the number of items in the collection for which the predicate is true. Give two versions, one using a loop and the other using the stream API.

See the Answers

Chapter 11

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** (pronounced "eye-oh"). 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 important, aside from files, is that it supports communication over network connections. In Java, the most common type of input/output involving files and networks is based on **I/O 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 I/O streams. (Note that I/O streams are **not** streams in the sense of the stream API that was covered in Section 10.6.)

Working with files and networks requires familiarity with exceptions, which were covered in Section 8.3. Many of the subroutines that are used can throw checked exceptions, which 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.

Contents of Chapter 11:

- Section 1: I/O Streams, Readers, and Writers
- Section 2: Files
- Section 3: Programming With Files
- Section 4: Networking
- Section 5: A Brief Introduction to XML
- Programming Exercises
- Quiz on This Chapter

Section 11.1

I/O 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 **I/O 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 several 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 the abstract class *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 the English 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 and characters in non-alphabetic written languages such as Chinese.)

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, since characters, like any other kind of data, have to be coded as binary numbers to be stored or processed by a computer. 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 prefer *Readers* and *Writers* rather than *InputStreams* and *OutputStreams* when working with character data, even when working with the standard ASCII character set.

The standard I/O 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. I/O streams are used 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.

(Note: The Java API provides additional support for I/O in the package `java.nio` and its subpackages, but they are not covered in this textbook. In general, `java.nio` gives programmers efficient access to more advanced I/O techniques.)

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 an abstract 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 a checked exception, 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. (Checked exceptions and mandatory exception handling were 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`, which can be a lot more efficient that reading individual 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 will rarely 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);
```

In fact, the parameter to the constructor can also be an *OutputStream* or a *File*, and the constructor will build a *PrintWriter* that can write to that output destination. (Files are covered in the next section.) When you output data to the *PrintWriter* `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 destination. 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 2.4.1, 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*.

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 ultimately 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. That 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 below, comes pretty close, but *Scanner* is not a subclass of any I/O stream class, which means that it doesn't fit neatly into the I/O stream framework. There is one basic

case that is easily handled by the standard class *BufferedReader*, which 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 MacOS, 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.)

BufferedReader also defines the instance method `lines()` which returns a value of type `Stream<String>` that can be used with the stream API (Section 10.6). A convenient way to process all the lines from a *BufferedReader*, `reader`, is to use the `forEachOrdered()` operator on the stream of lines: `reader.lines().forEachOrdered(action)`, where `action` is a consumer of strings, usually given as a lambda expression.

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`.

Note that the main purpose of *BufferedReader* is not simply to make it easier to read lines of text. Some I/O devices work most efficiently if data is read or written in large chunks, instead of as individual bytes or characters. A *BufferedReader* reads a chunk of data, and stores it in internal memory. The internal memory is known as a **buffer**. When you read from the BufferedReader, it will take data from the buffer if possible, and it will only go back to its input source for more data when the buffer is emptied. There is also a *BufferedWriter* class, and there are buffered stream classes for byte streams as well.

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 input source at a time and that it does not follow the same pattern as Java's built-in input/output classes. (If you like the style of input used by *TextIO*, you might take a look at my TextReader.java, which implements a similar style of input in a more object-oriented way. *TextReader* was used in previous versions of this textbook but is not used in this version.)

11.1.5 The Scanner Class

Since its introduction, Java has been notable for its lack of built-in support for basic input, or at least 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 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`. It was introduced in Subsection 2.4.6, but has not seen much use since then in this textbook. From now on, however, most of my examples will use *Scanner* instead of *TextIO*.

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*, among other possibilities. 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. 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, tabs, and end-of-line markers. 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 — read 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. The return value can be the empty string.

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 — return 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, *TextIO*, *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 *TextIO*-style input. *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.

(It is possible to change the delimiter that is used by a *Scanner*, but the syntax uses something called "regular expressions." Unfortunately, the syntax for regular expressions is rather complicated, and they are not covered in this book. However, as an example, suppose you want tokens to be words that consist entirely of letters of the English alphabet. In that case, delimiters should include all non-letter characters. If you want a *Scanner*, `scnr`, to use that kind of delimiter, you can say: `scnr.useDelimiter("[^a-zA-Z]+")`. After that, tokens returned by `scnr.next()` will consist entirely of letters. The string `"[^a-zA-Z]+"` is a regular expression. Regular expressions are an important tool for a working programmer. If you have a chance to learn about them, you should do so.)

11.1.6 Serialized Object I/O

The classes *PrintWriter*, *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 a lot of the work for you by using the classes *ObjectInputStream* and *ObjectOutputStream*. These are subclasses of *InputStream* and *OutputStream* that can be used for reading and writing 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. In the file, the primitive types will be represented in their internal binary format.

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 Section 11.5 for a character-based approach. See Subsection 12.5.1 for an example that uses object streams for network communication.)

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.

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. That is, the modified value will not be written correctly to the stream. 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*, and if it is possible that you will write the same object more than once, you can ensure that the full, correct version of the object will be written by calling the stream's `reset()` method before writing the object to the stream.

Section 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** (also 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 I/O streams. Human-readable character data can be read from a file using an object belonging to the class *FileReader*, which is a subclass of *Reader*. Similarly, data can be 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.

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. 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 *BufferedReader*, or in some other wrapper class. (See the previous section for a discussion of *BufferedReader* and *Scanner*.) To create a *BufferedReader* for reading from a file named `data.dat`, you could say:

```
BufferedReader data;

try {
   data = new BufferedReader( new FileReader("data.dat")
);
}
catch (FileNotFoundException e) {
   ... // handle the exception
}
```

Wrapping a *Reader* in a *BufferedReader* lets you easily read lines of text from the file, and the buffering can make the input more efficient.

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 more directly 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
}
```

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
}
```

However, as with *Scanner*, it is more common to use a constructor that takes a *File* as parameter; this will automatically wrap the File in a FileWriter before creating the PrintWriter:

```
PrintWriter result;

try {
   result = new PrintWriter(new File("result.dat"));
}
catch (IOException e) {
   ... // handle the exception
}
```

You can even use just a *String* as the parameter to the constructor, and it will be interpreted as a file name (but you should remember that a *String* in the *Scanner* constructor does not name a file; instead the scanner will read characters from the string itself).

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.

When you are finished with a *PrintWriter*, you should call its `flush()` method, such as "`result.flush()`", to make sure that all the output has been sent to its destination. If you forget to do this, you might find that some of the data that you have written to a file output stream has not actually shown up in the file.

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 *PrintWriter*, *BufferedReader*, 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 I/O stream. (Note that for most I/O stream classes, including *BufferedReader* the `close()` method can throw an *IOException*, which must be handled; however, *PrintWriter* 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 it is better not to depend on this. Note that calling `close()` should automatically call `flush()` before the file is closed. (I have seen that fail, but not recently.)

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 real numbers. The input file is read using a *Scanner*. 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.

```
import java.io.*;
import java.util.ArrayList;
import java.util.Scanner;

/**
 * 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
 * only real numbers.
 */
public class ReverseFileWithScanner {

    public static void main(String[] args) {

        Scanner data;        // For reading the 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 Scanner(new File("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("result.dat");
        }
        catch (FileNotFoundException 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.
```

```
        }

        while ( data.hasNextDouble() ) {  // Read until
end-of-file.
            double inputNumber = data.nextDouble();
            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!");

        data.close();
        result.close();

    }  // end of main()

} // end class ReverseFileWithScanner
```

Note that this program will simply stop reading data from the file if it encounters anything other than a number in the input. That will not be considered to be an error.

As mentioned at the end of Subsection 8.3.2, the pattern of creating or opening a "resource," using it, and then closing the resource is a very common one, and the pattern is supported by the syntax of the `try..catch` statement. Files are resources in this sense, as are *Scanner*, *PrintWriter*, and all of Java's I/O streams. All of these things define `close()` methods, and it is good form to close them when you are finished using them. Since they all implement the *AutoCloseable* interface, they are all resources in the sense required by `try..catch`. A `try..catch` statement can be used to automatically close a resource when the `try` statement ends, which eliminates the need to close it by hand in a `finally` clause. This assumes that you will open the resource and use it in the same `try..catch`.

As an example, the sample program ReverseFileWithResources.java is another version of the example we have been looking at. In this case, `try..catch` statements using the resource pattern are used to read the data

from a file and to write the data to a file. My original program opened a file in one `try` statement and used it in another `try` statement. The resource pattern requires that it all be done in one `try`, which requires some reorganization of the code (and can sometimes make it harder to determine the exact cause of an exception). Here is the `try..catch` statement from the sample program that opens the input file, reads from it, and closes it automatically.

```
try( Scanner data = new Scanner(new File("data.dat")) ) {
        // Read numbers, adding them to the ArrayList.
    while ( data.hasNextDouble() ) {  // Read until end-
of-file.
        double inputNumber = data.nextDouble();
        numbers.add( inputNumber );
    }
}
catch (FileNotFoundException e) {
        // Can be caused if file does not exist or can't
be read.
    System.out.println("Can't open input file data.dat!");
    System.out.println("Error: " + e);
    return;  // Return from main(), since an error has
occurred.
}
```

The resource, `data`, is constructed on the first line. The syntax requires a declaration of the resource with an initial value, in parentheses after the word "try." It's possible to have several resource declarations, separated by semicolons. They will be closed in the order opposite to the order in which they are declared.

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 MacOS 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.
- `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.dat." In general, "`..`" means "go up one directory." The corresponding path on Windows is `..\examples\data.dat`.

When working on the command line, it's safe to say 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 does not actually represent a file! Precisely speaking, 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 a directory. The second can be the name of the file in that directory or a relative path from that 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 output stream. The boolean function `file.canRead()` returns true if the file exists and the program has permission to read the file. And `file.canWrite()` is true if the program has permission to write to the file.
- `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`. The method `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. JavaFX includes a platform-independent technique for using file dialog boxes in the form of a class called *FileChooser*, in package `javafx.stage`.

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 constructor for *FileChooser* has no parameter. Constructing a *FileChooser* object does not make the dialog box appear on the screen. You have to call a method in the object to do that. Often, before showing the dialog box, you will call instance methods in the *FileChooser* object to set some properties of the dialog box. For example, you can set the file name that is shown to the user as a default initial value for the file.

A file dialog box can have an "owner," which is a window. In JavaFX, that means an object of type *Stage*. Until the dialog box is dismissed by the user —either by canceling the dialog or selecting a file—all interaction with the owner window is blocked. The owner can be specified as a parameter to the method that opens the dialog. The owner can be `null`, which will mean that no window is blocked.

There are two types of file dialog: an **open file dialog** that allows the user to specify an existing file to be opened for reading data into the program; and a **save file dialog** that lets the user specify a file, which might or might not already exist, to be opened for output. A *FileChooser* has two instance methods for showing the two kinds of dialog box on the screen. Suppose that `fileDialog` is a variable of type *FileChooser*. Then the following methods are available:

- `fileDialog.showOpenDialog(window)` — shows an open file dialog box on the screen. The parameter specifies the owner of the dialog box. This method does not return until the user has selected a file or canceled the dialog without selecting a file. The method returns a value of type *File*. The return value is `null` if the user canceled the dialog box. Otherwise, it returns a *File* object representing the selected file.
- `fileDialog.showSaveDialog(window)` — shows a save file dialog box, with owner equal to `window`. The parameter and return value are the same as for `showOpenDialog()`. If the user selects a file that already exists, the system will automatically ask whether the user wants to replace that file. So, you can safely go ahead and save the specified file.
- `fileDialog.setTitle(title)` — specifies a title to appear in the title bar of the dialog box. The parameter is a string. This method must

be called before showing the dialog box.

- `fileDialog.setInitialFileName(name)` — sets the name that appears as a default name in the input box for the file name. The parameter is a string. If the parameter is null, the input box will be empty; that is also the default. This method must be called before showing the dialog box.
- `fileDialog.setInitialDirectory(directory)` — sets the directory that is shown in the dialog box when it first appears. The parameter is of type *File*. If the parameter is `null`, the initial directory will be a system-dependent default (possibly the directory from which the program was run). If the parameter is not `null`, it must be a *File* object that represents a directory, not a regular file, or an error will occur. This method must be called before showing the dialog box.

A typical program has "Save" and "Open" commands for working with files. When the user selects a file for saving or opening, it can be a good idea to store the selected *File* object in an instance variable. Later, that file can be used to initialize the directory and possibly the file name the next time a file dialog box is created. If `editFile` is the instance variable that records the selected file, and if it is non-null, then `editFile.getName()` is a *String* giving the name of the file, and `editFile.getParent()` is a *File* representing the directory that contains the file.

This leaves open one question: what to do when an error occurs while reading or writing the selected file? The error should be caught, and the user should be informed that an error occurred. In a GUI program, the natural way to do that is with another dialog box that shows an error message to the user and has an "OK" button for dismissing the dialog. Dialog boxes were not covered in Chapter 6, but some common simple dialog boxes can be shown using objects of type *Alert*, from package `javafx.scene.control`. (See Subsection 13.4.1 for more about alerts.) Here is how to show an error message to the user:

```
Alert errorAlert = new Alert( Alert.AlertType.ERROR,
message );
errorAlert.showAndWait();
```

Putting all this together, we can look at a typical subroutine that saves data to a file. The file is selected using a *FileChooser*. In this example, the data is written in text form, using a *PrintWriter*:

```
private void doSave() {
    FileChooser fileDialog = new FileChooser();
    if (editFile == null) {
            // No file is being edited.  Set file name to
"filename.txt"
            // and set the directory to the user's home
directory.
        fileDialog.setInitialFileName("filename.txt");
        fileDialog.setInitialDirectory(
                new File( System.getProperty("user.home"))
);
    }
    else {
            // Get the file name and directory for the
dialog from
            //       the file that is currently being
edited.
        fileDialog.setInitialFileName(editFile.getName());

fileDialog.setInitialDirectory(editFile.getParentFile());
    }
    fileDialog.setTitle("Select File to be Saved");
    File selectedFile =
fileDialog.showSaveDialog(mainWindow);
    if ( selectedFile == null )
        return;  // User did not select a file.
    // Note: User has selected a file AND, if the file
exists, has
    //    confirmed that it is OK to erase the exiting
file.
    PrintWriter out;
    try {
        FileWriter stream = new FileWriter(selectedFile);
        out = new PrintWriter( stream );
    }
    catch (Exception e) {
            // Most likely, user doesn't have permission to
write the file.
        Alert errorAlert = new
Alert(Alert.AlertType.ERROR,
                "Sorry, but an error occurred while\n" +
                trying to open the file for output.");
        errorAlert.showAndWait();
```

```
        return;
    }
    try {
            .
            .   // WRITE TEXT TO THE FILE, using the
PrintWriter
            .
        out.flush(); // (not needed?; it's probably done
by out.close();
        out.close();
        if (out.checkError())   // (need to check for
errors in PrintWriter)
            throw new IOException("Error check failed.");
        editFile = selectedFile;
    }
    catch (Exception e) {
        Alert errorAlert = new
Alert(Alert.AlertType.ERROR,
                "Sorry, but an error occurred while\n" +
                "trying to write data to the file.");
        errorAlert.showAndWait();
    }
}
```

This general outline can easily be adapted to non-text files by using a different type of output stream.

Reading data from a file is similar, and I won't show the corresponding `doOpen()` method here. You can find working subroutines for saving and opening text files in the sample program TrivialEdit.java, which lets the user edit small text files. The file subroutines in that program can be adapted to many GUI programs that work with files.

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. Subsection 4.3.6 included a program for copying text files using *TextIO*. The example in this section will work for any file.

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 the byte streams, *InputStream* and *OutputStream*, to operate on the file. 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 4.3.6.) 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.

The program CopyFile.java 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 actually quite inefficient 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 data from and write data to files in larger blocks. This would require

changing only the two lines in the program that create the streams. For example, the input stream could be created using

```
source = new BufferedInputStream(new
FileInputStream(sourceName));
```

The buffered stream would then be used in exactly the same way as the unbuffered stream.

There is also a sample program CopyFileAsResources.java that does the same thing as `CopyFile` but uses the resource pattern in a `try..catch` statement to make sure that the streams are closed in all cases. See the discussion at the end of Subsection 8.3.2)

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 the file exists, it should contain the user's phone book, which was saved in a previous run of the program; in that case, 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" );
        // A file named .phone_book_data in the user's
home directory.

if ( ! dataFile.exists() ) {
   System.out.println("No phone book data file found.  A
new one");
   System.out.println("will be created, if you add any
entries.");
   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.flush();
   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.

11.3.3 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 7.3.3. (You might want to run it now to remind yourself of what it can do.) In that program, the user can 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 implements "Save" and "Open" commands for writing program data to a file and reading saved data back into the program.

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 *Point2Ds*. A *Point2D*, `pt`, has instance methods `pt.getX()` and `pt.getY()` that return the coordinates of a point in the xy-plane as values of type double. 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

are 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 {
   Color color;  // The color of the curve.
   boolean symmetric;  // Are horizontal and vertical
reflections also drawn?
   ArrayList<Point2D> points;  // The points on the curve.
}
```

Then, a list of type `ArrayList<CurveData>` is used to hold data for all of the curves that the user has drawn.

Let's think about how the data for a sketch could be saved to a text file. The basic idea is that all 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, all of 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 numbers 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:

```
out.println( backgroundColor.getRed() ); // Write
background color to file.
out.println( backgroundColor.getGreen() );
out.println( backgroundColor.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 ( Point2D pt : curve.points ) {       // the
coordinates of each point
      out.println( pt.getX() );
      out.println( pt.getY() );
   }
}
```

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.
double red = scanner.nextDouble();
double green = scanner.nextDouble();
double blue = scanner.nextDouble();
newBackgroundColor = Color.color(red,green,blue);

ArrayList<CurveData> newCurves = new ArrayList<>();

int curveCount = scanner.nextInt();      // The number of
curves to be read.
for (int i = 0; i < curveCount; i++) {
   CurveData curve = new CurveData();
   double r = scanner.nextDouble();            // Read the
curve's color.
   double g = scanner.nextDouble();
   double b = scanner.nextDouble();
   curve.color = Color.color(r,g,b);
   int symmetryCode = scanner.nextInt(); // Read the
curve's symmetry property.
   curve.symmetric = (symmetryCode == 1);
   curveData.points = new ArrayList<>();
   int pointCount = scanner.nextInt();  // The number of
points on this curve.
   for (int j = 0; j < pointCount; j++) {
      int x = scanner.nextDouble();        // Read the
```

```
coordinates of the point.
      int y = scanner.nextDouble();
      curveData.points.add(new Point2D(x,y));
   }
   newCurves.add(curve);
}

curves = newCurves;                      // Install the new
data structures.
backgroundColor = 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 0.4 0.4 0.5

startcurve
  color 1 1 1
  symmetry true
  coords 10 10
  coords 200 250
  coords 300 10
endcurve

startcurve
  color 0 1 1
  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 numbers 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 with differing line widths. 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 is be done with:

```
out.println("SimplePaintWithFiles 1.0"); // Version
number.
out.println( "background " + backgroundColor.getRed() + "
" +
        backgroundColor.getGreen() + " " +
backgroundColor.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 ( Point2D pt : curve.points )
```

```
        out.println( "  coords " + pt.getX() + " " +
pt.getY() );
    out.println("endcurve");
}
```

In the program, this code is used in a `doSave()` method that is similar to the one that is presented in Subsection 11.2.3. The method uses a file dialog box to allow the user to select the output file.

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 created by the `doSave()` method. It uses a *Scanner* to read items from the file:

```
private void doOpen() {
    FileChooser fileDialog = new FileChooser();
    fileDialog.setTitle("Select File to be Opened");
    fileDialog.setInitialFileName(null);  // No file is
initially selected.
    if (editFile == null)
        fileDialog.setInitialDirectory(new
File(System.getProperty("user.home")));
    else

fileDialog.setInitialDirectory(editFile.getParentFile());
    File selectedFile = fileDialog.showOpenDialog(window);
    if (selectedFile == null)
        return;  // User canceled.
    Scanner scanner;
    try {
        scanner = new Scanner( selectedFile );
    }
    catch (Exception e) {
        Alert errorAlert = new
Alert(Alert.AlertType.ERROR,
                "Sorry, but an error occurred\nwhile
trying to open the file.");
        errorAlert.showAndWait();
```

```
        return;
    }
    try {
        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 a newer
version of SimplePaintWithFiles.");
        Color newBackgroundColor = Color.WHITE;
        ArrayList<CurveData> newCurves = new
ArrayList<CurveData>();
        while (scanner.hasNext()) {
            String itemName = scanner.next();
            if (itemName.equalsIgnoreCase("background")) {
                double red = scanner.nextDouble();
                double green = scanner.nextDouble();
                double blue = scanner.nextDouble();
                newBackgroundColor =
Color.color(red,green,blue);
            }
            else if
(itemName.equalsIgnoreCase("startcurve")) {
                CurveData curve = new CurveData();
                curve.color = Color.BLACK;
                curve.symmetric = false;
                curve.points = new ArrayList<Point2D>();
                itemName = scanner.next();
                while ( !
itemName.equalsIgnoreCase("endcurve") ) {
                    if
(itemName.equalsIgnoreCase("color")) {
                        double r = scanner.nextDouble();
                        double g = scanner.nextDouble();
                        double b = scanner.nextDouble();
                        curve.color = Color.color(r,g,b);
                    }
                    else if
(itemName.equalsIgnoreCase("symmetry")) {
                        curve.symmetric =
scanner.nextBoolean();
                    }
                    else if
(itemName.equalsIgnoreCase("coords")) {
                        double x = scanner.nextDouble();
```

```
                        double y = scanner.nextDouble();
                        curve.points.add( new Point2D(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();
        backgroundColor = newBackgroundColor;
        curves = newCurves;
        redraw();
        editFile = selectedFile;
        window.setTitle("SimplePaint: " +
editFile.getName());
    }
    catch (Exception e) {
        Alert errorAlert = new
Alert(Alert.AlertType.ERROR,
                "Sorry, but an error occurred
while\ntrying to read the data:\n"
                        + e);
        errorAlert.showAndWait();
    }
}
```

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 text 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 Subsection 13.2.6, and it uses techniques that I have not yet covered.

Section 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. A *URLConnection* represents a network connection to such a resource.

The second style of I/O, which is more general and 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—or even running on the same computer. 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." (URLs can actually refer to resources from other sources besides the web; after all, they are "universal." For example, they can represent files on your computer.)

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 "`https://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 `https://math.hws.edu/eck/`, then the incomplete, relative url "`index.html`" would really refer to `https://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, plus 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.

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/png", 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 "png". 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.
Note:
        connection.getContentType() method should be
called
        after getInputStream(). */

     String contentType = connection.getContentType();
     System.out.println("Stream opened with content type:
" + contentType);
     System.out.println();
     if (contentType == null ||
contentType.startsWith("text") == false)
          throw new IOException("URL does not seem to
refer to a text file.");
     System.out.println("Fetching content from " +
urlString + " ...");
     System.out.println();

     /* 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);
     }
     in.close();

} // end readTextFromURL()
```

A complete program that uses this subroutine can be found in the file FetchURL.java. When you run the program, you can specify the URL on the command line; if not, you will be prompted to enter the URL. For this program, a URL can begin with "http://" or "https://" for a URL that refers to a resource on the web, with "file://" for a URL that refers to a file on your computer, or with "ftp://" for a URL that uses the "File Transfer Protocol." If it does not start with any of these, then "http://" is added to the start of the URL. Try the program with URL `math.hws.edu/javanotes` to fetch the front page of this textbook from its website. Try it with some bad inputs to see the various errors that can occur.

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 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 the listening socket in a server of that type, that socket does not stop listening. Instead, it continues listening for additional client connection requests at the same time that the client is being serviced. But to allow multiple clients to be connected at the same time, 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. 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 `64.89.144.237`, 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 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). You might see IP addresses of either form.

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 one computer, the output was:

```
   wlo1 :  /2603:7080:7f41:f400:441a:459f:2c62:c349%wlo1
/192.168.0.2
   lo :  /0:0:0:0:0:0:0:1%lo  /127.0.0.1
```

The first thing on each line is a network interface name, which is really meaningful only to the computer's operating system. The same line also contains the IP addresses for that interface. In this example, `lo` refers to the loopback address, which has IPv4 address 127.0.0.1 as usual. The most important number here is `192.168.0.2`, which is the IPv4 address that can

be used for communication over the network. (The slashes at the start of each address are not part of the actual address.) The other numbers in the output are IPv6 addresses, but the IPv4 addresses are easier for a human to use.

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 actually has a **port number** in combination with an IP address. A port number is just a 16-bit positive 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 port number smaller than 1024 is specified. An *IOException* can also 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 that you want it to continue to accept connections as long as the program is running. 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();
   }
   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. 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 can be specified as a command-line argument. For example, if the server is running on a computer named math.hws.edu, then you could run the client with the command "`java DateClient math.hws.edu`". If a computer is not

specified on the command line, then the user is prompted to enter one. Here is the complete client program:

```
import java.net.*;
import java.util.Scanner;
import java.io.*;

/**
 * This program opens a connection to a computer specified
 * as the first command-line argument.  If no command-line
 * argument is given, it prompts the user for a computer
 * to connect to.  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 server.
        BufferedReader incoming; // For reading data from
the connection.

        /* Get computer name from command line. */

        if (args.length > 0)
            hostName = args[0];
        else {
            Scanner stdin = new Scanner(System.in);
            System.out.print("Enter computer name or IP
address: ");
            hostName = stdin.nextLine();
        }

        /* 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. If that method returns `null`, it probably means that the server closed the connection without sending any data.

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 would 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 reasonably 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 environment 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.util.Scanner;
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.

        Scanner 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();
            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 Scanner(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.nextLine();
                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. If the server is not listening on the default port, you can add the port number as a second command line option to the program. If you do not specify the connection information on the command line, you will be asked to enter it.

Section 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 are also alternatives to DTDs, such as **XML schemas**, for defining valid XML 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 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

If you know HTML, the language for writing web pages, then XML will look familiar. An XML document looks a lot like an HTML document. 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='1' green='0.6' blue='0.2'/>
   <curve>
      <color red='0' green='0' blue='1'/>
      <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='1' green='1' blue='1'/>
      <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 author of a well-formed XML document gets to choose the tag names and attribute names, and meaningful names can be chosen to describe the data to a human reader. (For a valid 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 `>`, `"`, and `'` 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 Working With the DOM

The sample XML file shown above was designed to store information about simple drawings made by the user. The drawings in question are ones that could be made using the sample program SimplePaint2.java from Subsection 7.3.3. We'll look at another version of that program that can save the user's drawing using an XML format for the data file. The new version is SimplePaintWithXML.java. The sample XML document shown earlier in this section can be used with that program. I designed the format of that document to represent all the data needed to reconstruct a picture in `SimplePaint2`. 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 `SimplePaintWithXML` writes the data for a `SimplePaint2` picture to a *PrintWriter*, `out`. This produces an XML file with the same structure as the example shown above:

```
out.println("<?xml version=\"1.0\"?>");
out.println("<simplepaint version=\"1.0\">");
out.println("   <background red='" +
backgroundColor.getRed() + "' green='" +
        backgroundColor.getGreen() + "' blue='" +
backgroundColor.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 (Point2D pt : c.points)
        out.println("      <point x='" + pt.getX() + "'
y='" + pt.getY() + "'/>");
    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 well-formed 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 for processing XML 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, and `xmldoc` is of type *Document*, 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 of 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

```
<background red='1' green='0.6' blue='0.2'/>
```

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:

```
double r = Double.parseDouble( element.getAttribute("red")
);
double g = Double.parseDouble(
element.getAttribute("green") );
double b = Double.parseDouble(
element.getAttribute("blue") );
Color bgColor = Color.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 *Point2Ds*:

```
<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<Point2D>` that is represented by the element. We can do this by traversing the *NodeList* that contains the child nodes of `element`:

```
ArrayList<Point2D> points = new ArrayList<>();
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.
      double x = Double.parseDouble(
pointElement.getAttribute("x") );
      double y = Double.parseDouble(
pointElement.getAttribute("y") );
      Point2D pt = new Point2D(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<>();
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")) {
         double r =
Double.parseDouble(element.getAttribute("red"));
         double g =
Double.parseDouble(element.getAttribute("green"));
         double b =
Double.parseDouble(element.getAttribute("blue"));
         newBackground = Color.color(r,g,b);
      }
      else if (element.getTagName().equals("curve")) {
         CurveData curve = new CurveData();
         curve.color = Color.BLACK;
         curve.points = new ArrayList<>();
         newCurves.add(curve);
         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")) {
               double r =
Double.parseDouble(curveElement.getAttribute("red"));
               double g =
Double.parseDouble(curveElement.getAttribute("green"));
               double b =
Double.parseDouble(curveElement.getAttribute("blue"));
               curve.color = Color.color(r,g,b);
             }
             else if
(curveElement.getTagName().equals("point")) {
               double x =
Double.parseDouble(curveElement.getAttribute("x"));
               double y =
Double.parseDouble(curveElement.getAttribute("y"));
               curve.points.add(new Point2D(x,y));
             }
             else if
(curveElement.getTagName().equals("symmetric")) {
               String content =
curveElement.getTextContent();
               if (content.equals("true"))
                 curve.symmetric = true;
```

```
                }
            }
        }
    }
}
}
backgroundColor = newBackground;
curves = newCurves;
```

You can find the complete source code in SimplePaintWithXML.java.

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.

Programming Exercises for Chapter 11

Exercise 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.
```

See the Solution

Exercise 11.2:

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* or a *BufferedReader* to process each file.

See the Solution

Exercise 11.3:

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 a line beginning with the word "ERROR" to the client and closes the connection. (The error response can include an error message on the rest of the line.)

Your program should use a subroutine to handle each request that the server receives. It should not stop after handling one request; it should remain open and continue to accept new requests. See the DirectoryList example in Subsection 11.2.2 for help with the problem of getting the list of files in the directory.

See the Solution

Exercise 11.4:

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).

See the Solution

Exercise 11.5:

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 11.5.2. 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 a short exercise in basic XML processing; as before, the program in this exercise is not meant to be a useful phone directory program.)

See the Solution

Exercise 11.6:

The sample program Checkers.java from Subsection 7.6.3 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, showing that Load and Save buttons have been added:

It's a little tricky to restore the complete state of a game after reading the data from a file. 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 remaining program state correctly. This code assumes that you have introduced two new variables `saveButton` and `loadButton` of type *Button* 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.setDisable(true);
loadButton.setDisable(true);
saveButton.setDisable(false);
resignButton.setDisable(false);
if (currentPlayer == CheckersData.RED)
   message.setText("Game loaded -- it's RED's move.");
else
```

```
   message.setText("Game loaded -- it's BLACK's move.");
drawBoard();
```

See the Solution

Quiz on Chapter 11

Question 1:

In Java, input/output is done using I/O streams. I/O streams are an *abstraction.* Explain what this means and why it is important.

Question 2:

Java has two types of I/O stream: character streams and byte streams. Why? What is the difference between the two types of streams?

Question 3:

What is a *file?* Why are files necessary?

Question 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*?

Question 5:

The package `java.io` includes a class named *URL*. What does an object of type *URL* represent, and how is it used?

Question 6:

What is the purpose of the *FileChooser* class?

Question 7:

Explain what is meant by the *client / server* model of network communication.

Question 8:

What is a *socket*?

Question 9:

What is a *ServerSocket* and how is it used?

Question 10:

What is meant by an *element* in an XML document?

Question 11:

What is it about XML that makes it suitable for representing almost any type of data?

Question 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; read from the file using methods covered in this chapter.

See the Answers

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.

As Java has developed, new features have been added to the language that make it possible to do parallel programming without using threads directly. We have already seen one of these: parallel streams in the stream API from Section 10.6. We will encounter several more in this chapter. But to use these higher level language features safely and efficiently, you still need to know something about the hazards and benefits of threads.

Contents of Chapter 12:

- Section 1: Introduction to Threads
- Section 2: Programming with Threads
- Section 3: Threads and Parallel Processing
- Section 4: Threads and Networking
- Section 5: Network Programming Example: A Networked Game Framework
- Programming Exercises
- Quiz on This Chapter

Section 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, most computers, and even mobile devices, now 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. In JavaFX, that thread is the application thread.

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 a variety of standard classes to help with some of the more tricky parts or to hide them entirely. 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. In fact, a programmer can do a lot with threads without ever learning about the low-level stuff.

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()` method 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 mostly 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 defining the thread using a nested class that is a subclass of the *Thread* class. (See Section 5.8.)

Finally, I'll note that *Runnable* is a functional interface, so that a *Runnable* object can be given as a lambda expression. In particular, the parameter to `new Thread(r)` can be a lambda expression. For example:

```
Thread greetingsFromFred = new Thread(
    () -> 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 10,000,000 that are prime. (The particular task that is done is not important for our purposes here, as long as it is something that takes a non-trivial amount of time. This is a demo program; it would be silly in a real program to have multiple threads that do the same thing, and the method that is used for counting the primes is very inefficient.) This computation should take no more than a few seconds on a modern computer. The threads that perform this task are defined by the following static nested class, where `MAX` is 10,000,000:

```
/**
 * When a thread belonging to this class is run it will
count the
 * number of primes between 2 and MAX.  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,MAX); // 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 > 30) {
      System.out.print("How many threads do you want to
use  (1 to 30) ?  ");
      numberOfThreads = TextIO.getlnInt();
      if (numberOfThreads < 1 || numberOfThreads > 30)
         System.out.println("Please enter a number between
1 and 30 !");
   }
   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.

My computer has six processors. When I ran the program on that computer with using one thread, it took about 3.36 seconds. When I ran it using twelve

threads, the output was:

```
Creating 12 prime-counting threads...
Threads have been created and started.
Thread 1 counted 664579 primes in 6.424 seconds.
Thread 11 counted 664579 primes in 6.694 seconds.
Thread 0 counted 664579 primes in 6.727 seconds.
Thread 3 counted 664579 primes in 6.74 seconds.
Thread 10 counted 664579 primes in 6.759 seconds.
Thread 7 counted 664579 primes in 6.77 seconds.
Thread 2 counted 664579 primes in 6.798 seconds.
Thread 9 counted 664579 primes in 6.792 seconds.
Thread 5 counted 664579 primes in 6.824 seconds.
Thread 4 counted 664579 primes in 6.836 seconds.
Thread 8 counted 664579 primes in 6.841 seconds.
Thread 6 counted 664579 primes in 6.877 seconds.
```

The second line was printed immediately after the first. At this point, the main program has ended but the twelve threads continued to run. After a pause of about 6.5 seconds, all eight 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, twelve threads took about twice as long as one thread. This is because the computer has six processors. With twelve threads running on six processors—one-half processor per thread—each thread was only actively being executed for about half of the time, so it took twice as long to finish its task. On a single-processor computer, twelve threads would take about twelve times as long as 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 any `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 (or until the waiting thread is interrupted). 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. (A call to `System.exit()` forces the JVM to terminate, even if there are non-daemon threads still running.)

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` to the specified value.

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 each of several threads performs 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 before another thread is allowed to access `count`. 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.

However, the *ThreadSafeCounter* class does not prevent all possible race conditions that might arise when using a counter. Consider the `if` statement:

```
if ( tsc.getValue() == 10 ) {
   doSomething();
}
```

where `doSomething()` is some method that requires the value of the counter to be 10. 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() == 10` 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() == 10 )
      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 means that it is not only true that two threads cannot be executing the same synchronized method at the same time, but in fact two threads cannot be executing two different methods at the same time, if the two methods are synchronized on the same 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.

The discussion of invariants in Subsection 8.2.3 mentioned that reasoning about invariants becomes much more complicated when threads are involved. The problem is race conditions. We would like our *ThreadSafeCounter* class to have the class invariant that "the value of `count` is the number of times that `increment()` has been called." In a single-threaded program, that would be true even without synchronization.

However, in a multithreaded program, synchronization is needed to ensure that the class invariant is really invariant.

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, and it is the only method in which the value of `total` is changed:

```
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 6,000,001 to 12,000,000. (The numbers are rather arbitrary.) The `main()` routine in this program creates between 1 and 6 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.

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 two such techniques: volatile variables and atomic 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.

It is safe to use a shared variable in a synchronized method or statement, as long as all access to that variable is synchronized, using the same synchronization object in all cases. More precisely, any thread that accesses a variable in synchronized code is guaranteed to see changes that were made by other threads, as long as the changes were made in code that was synchronized on the same object.

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 global 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 using a volatile variable does not solve race conditions that occur, for example, when the value of the variable is incremented. The increment operation can still be interrupted by another thread.)

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 or immutable types like *String*.

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 the main memory that is shared by all processors, 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 main memory.)

12.1.5 Atomic Variables

The problem with a statement like "`count = count + 1`" for parallel programming is that it takes several steps to execute the statement. The statement is only correctly executed if those steps are completed without interruption.

An **atomic** operation is something that cannot be interrupted. It is an all-or-nothing affair. It cannot be partly completed. Most computers have operations that are atomic on the machine language level. For example, there might be a machine language instruction that can atomically increment the value in a

memory location. Such an instruction could be used without fear of race conditions.

In a program, an operation can be atomic even if it is not literally atomic on the machine language level. An operation can be considered atomic if it can never be **seen** by any thread as partly completed. For example, we can say that the *ThreadSafeCounter* class that was defined above has an atomic increment operation. Synchronization can be seen as a way of ensuring that operations are, at least effectively, atomic.

Still, it would be nice to have atomic operations that don't require synchronization, especially since such operations might be implemented very efficiently on the hardware level.

Java has a package `java.util.concurrent.atomic` that defines classes that implement atomic operations on several simple variable types. We will look at the class *AtomicInteger*, which defines some atomic operations on an integer value, including atomic add, increment, and decrement. Suppose, for example, that we want to add some integer values that are being produced by a number of different threads. We can do that with an *AtomicInteger* such as

```
private static AtomicInteger total = new AtomicInteger();
```

The total is created with an initial value of zero. When a thread wants to add a value to the total, it can use the method `total.addAndGet(x)`, which adds `x` to the total and returns the new value of total after `x` has been added. This is an atomic operation, which cannot be interrupted, so we can be sure that the value of total will be correct in the end. The sample program ThreadTest3.java is a small variation on ThreadTest2.java that uses an *AtomicInteger* instead of synchronization to safely add up values from several threads.

AtomicInteger has similar methods for adding one to the total and subtracting one from the total: `total.incrementAndGet()` and `total.decrementAndGet()`. The method `total.getAndSet(x)` sets the value of the total to `x` and returns the previous value that `x` is replacing. All of these operations are done atomically (either because they use atomic

machine language instructions or because they use synchronization internally).

I should close with my usual warning: Using an atomic variable does not automatically solve all race conditions involving that variable. For example, in the code

```
int currentTotal = total.addAndGet(x);
System.out.println("Current total is " + currentTotal);
```

it is possible that by the time the output statement is executed, the total has been changed by another thread so that `currentTotal` is no longer the current value of total!

Section 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 periodic task at set intervals. In fact, this is so basic that there are specialized classes for performing this task—and you've already worked with one of them: the *AnimationTimer* class, in package `javafx.animation`, which was introduced in Subsection 6.3.5. An AnimationTimer calls its `handle()` method periodically, and you can program an animation by overriding that method in a subclass of *AnimationTimer*. Before timers were introduced, threads had to be used to implement a similar functionality.

Suppose that we wanted to do something similar with a thread. That is, we would like to call some subroutine at periodic intervals, say every 30 milliseconds. The `run()` method of the thread would have to execute a loop in which the thread sleeps for 30 milliseconds, then wakes up to call the subroutine. 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 MyAnimator extends Thread {
    public void run() {
       while (true) {
           try {
               Thread.sleep(30);
           }
           catch (InterruptedException e) {
```

```
            }
            callSubroutine();
        }
    }
}
```

To use this class, you would create an object belonging to it and call its `start()` method. As it stands, there would be no way to stop the thread once it is started. One way to make that possible would be to end the loop when a `volatile` boolean variable, `terminate`, becomes `true`, as discussed in Subsection 12.1.4. A thread object can only be executed once, so in order to restart the animation after it has been stopped in this way, 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 JavaFX *AniamtionTimer* does nothing but call its `handle()` routine repeatedly. That method is actually executed in the JavaFX application thread, which also handles repainting of components and responses to user actions. This is important because JavaFX 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 application thread, there is no problem. A problem can arise when another thread tries to manipulate components or the variables that are also used in the GUI thread. Using synchronization would be a solution in some cases. The best solution is probably to use an *AnimationTimer*, if that is possible. But if you really need to use a separate thread, you might be able to use `Platform.runLater()`.

`Platform.runLater(r)` is a static method in class *Platform*, from package `javafx.application`. The parameter is an object of type *Runnable*, the same interface that is used when creating threads. `Platform.runLater(r)` can be called from any thread. Its purpose is to submit `r` to be run on the JavaFX application thread. `Platform.runLater(r)` returns immediately, without waiting for `r` to be run; `r.run()` will be called by the application thread at some future time (which means within a fraction of a second, and perhaps almost immediately, unless the computer is excessively busy). *Runnables* are executed in the order in which they are submitted. Since the *Runnable* is called on the application thread, it can safely operate on the

GUI, without synchronization. It is often convenient to specify the parameter to `Platform.runLater()` as a lambda expression of type *Runnable*. I will use `Platform.runLater()` in several examples in this chapter and the next.

As an example, the sample program RandomArtWithThreads.java uses a thread to drive a very simple animation. In this example, the thread does nothing except to call a `redraw()` method every two seconds. The method redraws the content of a canvas. `Platform.runLater()` is used to execute `redraw()` on the application thread. The user can click a button to start and stop the animation. A new thread is created each time the animation is started. A `volatile` boolean variable, `running`, is set to `false` when the user stops the animation, as a signal to the thread to stop, as discussed in Subsection 12.1.4. The thread is defined by the following class:

```
private class Runner extends Thread {
    public void run() {
        while (running) {
            Platform.runLater( () -> redraw() );
            try {
                Thread.sleep(2000);  // Wait two seconds
between redraws.
            }
            catch (InterruptedException e) {
            }
        }
    }
}
```

12.2.2 Recursion in a Thread

One reason to use a separate thread to drive an animation is when the thread is running a recursive algorithm, and you want to redraw the display many times over the course of the recursion. (Recursion is covered in Section 9.1.) It's difficult to break up a recursive algorithm into a series of method calls in 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. In the program, the user clicks a "Start" button to start the process. The order of the colors is randomized and QuickSort is called to sort them, showing the process in slow motion. During the sort, the "Start" button changes to a "Finish" button that can be used to abort the sort before it finishes on its own. (It's an interesting program to watch, and it might even help you to understand QuickSort better, so you should try running it.)

In this program, the picture in a *Canvas* needs to change every time the algorithm makes a change to the array. The array is changed in the animation thread, but the corresponding change to the canvas has to be made in the JavaFX application thread, using `Platform.runLater()`, as discussed above. Each time it calls `Platform.runLater()`, the animation thread sleeps for 100 milliseconds to allow time for the application thread to run the `Runnable` 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.

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();
```

```
   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 thread that runs the recursive
 * QuickSort algorithm.  The thread begins by randomizing
the
 * hue array.  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.  In any case, in the end, it
 * resets the text on the button to "Start".
 */
private class Runner extends Thread {
    public void run() {
        for (int i = 0; i < hue.length; i++) {
               // fill hue array with indices in order
            hue[i] = i;
        }
        for (int i = hue.length-1; i > 0; i--) {
               // Randomize the order of the hues.
            int r = (int)((i+1)*Math.random());
            int temp = hue[r];
            hue[r] = hue[i];
               // The last assignment that needs to be
done in this
               // loop is hue[i] = temp.  The value of
hue[i] will
               // not change after this, so the assignment
is done
               // by calling a method setHue(i,temp) that
```

```
will change
                // the value in the array and also use
Platform.runLater()
                // to change the color of the i-th color
bar in the canvas.
            setHue(i,temp);
        }
        try {
            delay(1000);  // Wait one second before
starting the sort.
            quickSort(0,hue.length-1);  // Sort the whole
array.
        }
        catch (ThreadTerminationException e) { // User
aborted quickSort.
                // Put the colors back into sorted order.
The drawSorted()
                // method draws all of the color bars in
sorted order.
            Platform.runLater( () -> drawSorted() );
        }
        finally {
            running = false;  // make sure running is
false; this is only
                              //   really necessary if the
thread terminated
                              //   normally
            Platform.runLater( () ->
startButton.setText("Start") );
        }
    }
}
```

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.

When the user clicks the "Finish" button, the value of `running` is set to `false` as a signal to the thread to terminate. However, the thread might be sleeping when the "Finish" button is pressed, and in that case, the thread has to wake up before it can act on the signal. To make the thread a little more responsive, the program calls `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 (or in *Runnables* passed to `Platform.runlater()`). A 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 then arrange for that frame to be drawn. This leaves plenty of time while the animation thread is sleeping between frames for the JavaFX application thread to do any necessary redrawing of the GUI and to handle any other events.

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—redrawing 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 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 a while to compute because 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.

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 uses `Platform.runLater() to` copy 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 JavaFX application 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();
```

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. (And in fact, in that case the business about thread priority is not all that relevant, because the animation thread and the application thread can run both run simultaneously, on different processors.) It would be nice to get **all** of the processors working on the computation. 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 a picture of the program nearing the end 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. 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, using `Platform.runLater()`.

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

certainly can't do that in the JavaFX application thread!) In this example, I use an instance variable, `threadsRunning`, to keep track of how many computation threads are still running. As each thread finishes, it calls a method that subtracts one from 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 thread ends.) When the number of running threads drops to zero, the method updates the state of the program appropriately. Here is the method that is called by each thread just before terminating:

```
synchronized private void threadFinished() {
    threadsRunning--;
    if (threadsRunning == 0) { // all threads have
finished
        Platform.runLater( () -> {
               // make sure GUI state is correct when
threads end.
            startButton.setText("Start Again");
            startButton.setDisable(false);
            threadCountSelect.setDisable(false);
        });
        running = false; // Make sure running is false
after the threads end.
        workers = null;  // Discard the array that holds
the threads.
    }
}
```

Note that this method is `synchronized`. This is to avoid the race condition when `threadsRunning` is decremented. 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 `threadsRunning` and get the same answer when they decrement it. The net result will be that `threadsRunning` goes down by one instead of by two. One thread is not properly counted, and `threadsRunning` will never become zero. 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.

Section 12.3

Threads and Parallel Processing

THE EXAMPLE IN Subsection 12.2.4 in 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 a little 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 computes 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. But for the most part, that's a topic for a course in parallel computing, not an introductory programming course.

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; to create a queue that can hold objects of type *Runnable*, you can say

```
ConcurrentLinkedQueue<Runnable> queue = new
ConcurrentLinkedQueue<>();
```

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! On the other hand, `queue.poll()` is an "atomic" operation (Subsection 12.1.5).

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`, even if there are still tasks remaining in the queue. The threads are defined by a nested class named *WorkerThread*. It is quite short and simple to write:

```
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.
                              // Done in finally to make
sure it gets called.
        }
    }
}
```

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, that is, 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.
for (int row = 0; row < height; row++) {  // height is
number of rows in image
    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
(selected by user).
workers = new WorkerThread[threadCount];
running = true;  // Set the signal before starting the
threads!
threadsRemaining = workers;  // Records how many threads
are still running.
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 started, they might see an empty queue and terminate immediately after being started, without performing any tasks!

You should try out `MultiprocessingDemo2`. It computes the same image as `MultiprocessingDemo1`, but the rows of pixels are not computed in the same order as in that program (if 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. Try it with 20 threads.)

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 place items into the queue as they are produced. Consumers 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 most appropriate for use with blocking queues that have a limited capacity; `bqueue.add(item)` does not block and is most appropriate 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<>(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 items 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<>();
```

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 of producers, 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 in this program 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
automatically when
 * the only threads are daemon threads, so the existence
of the thread
 * pool will not stop the JVM from exiting.
 */
private class WorkerThread extends Thread {
    WorkerThread() {
        try {
            setPriority(
Thread.currentThread().getPriority() - 1);
        }
        catch (Exception e) {
        }
        try {
```

```
            setDaemon(true);
        }
        catch (Exception e) {
        }
        start(); // Thread starts as soon as it is
constructed.
    }
    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.

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 `running` 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 The ExecutorService Approach

Since thread pools are common in parallel programming, it is not surprising that Java has higher level tools for creating and managing thread pools. The interface *ExecutorService*, in package `java.util.concurrent`, defines services that can execute tasks that are submitted to it. Class *Executors* contains `static` methods that can be used to create *ExecutorServices* of various types. In particular, `Executors.newFixedThreadPool(n)`, where `n` is an int, creates a thread pool with `n` threads. To get a thread pool with one thread per available processor, you can say

```
int processors =
Runtime.getRuntime().availableProcessors();
ExecutorService executor =
Executors.newFixedThreadPool(processors);
```

The method `executor.execute(task)` can be used to submit a *Runnable* object, `task`, for execution. The method returns immediately after placing `task` into a queue of waiting tasks. Threads in the thread pool remove tasks from the queue and execute them.

The method `executor.shutdown()` tells the thread pool to shut down after all waiting tasks have been executed. The method returns immediately, without waiting for the threads to finish. After this method has been called, it is not legal to add new tasks. It is **not** an error to call `shutdown()` more than once. The threads in the thread pool are not daemon threads; if the service is not shut down, the existence of the threads will stop the Java Virtual Machine

from shutting down after other threads have exited. Before exiting, the threads in the pool will complete any tasks they have already removed from the queue.

The method `executor.shutdownNow()` is similar to `executor.shutdown()` but it discards any waiting tasks that are still in the queue, and tries to stop any tasks that are currently being executed.

The sample program MultiprocessingDemo4.java is a variation on `MultiprocessingDemo3` that uses an *ExecutorService* instead of using threads and a blocking queue directly. (Since I have not found an easy way to get an `ExecutorService` to drop waiting tasks without shutting down, `MultiprocessingDemo4` creates a new `ExecutorService` each time it computes a new image.)

Tasks for an *ExecutorService* can also be represented by objects of type `Callable<T>`, which is a parameterized functional interface that defines the method `call()` with no parameters and a return type of *T*. A *Callable* represents a task that outputs a value.

A *Callable*, `c`, can be submitted to an *ExecutorService* by calling `executor.submit(c)`. The `Callable` will then be executed at some future time. The problem is, how to get the result of the computation when it completes? This problem is solved by another interface, `Future<T>`, which represents a value of type *T* that might not be available until some future time. The method `executor.submit(c)` returns a *Future* that represents the result of the future computation.

A *Future*, `v`, defines several methods, including `v.isDone()`, which is a boolean-valued function that can be called to check whether the result is available; and `v.get()`, which will retrieve the value of the future. The method `v.get()` will block until the value is available. It can also generate exceptions and needs to be called in a `try..catch` statement.

As an example, ThreadTest4.java uses *Callables*, *Futures*, and an *ExecutorService* to count the number of primes in a certain range of integers. (This is the same rather useless computation that was done by

ThreadTest2.java in Subsection 12.1.3.) In this program, a subtask counts the primes in a subrange of integers. The subtasks are represented by objects of type `Callable<Integer>`, defined by this nested class:

```
/**
 * An object belonging to this class will count primes in
a specified range
 * of integers.  The range is from min to max, inclusive,
where min and max
 * are given as parameters to the constructor.  The
counting is done in
 * the call() method, which returns the number of primes
that were found.
 */
private static class CountPrimesTask implements
Callable<Integer> {
    int min, max;
    public CountPrimesTask(int min, int max) {
        this.min = min;
        this.max = max;
    }
    public Integer call() {
        int count = countPrimes(min,max);  // does the
counting
        return count;
    }
}
```

All the subtasks are submitted to a thread pool implemented as an *ExecutorService*, and the *Futures* that are returned are saved in an array list. In outline:

```
int processors =
Runtime.getRuntime().availableProcessors();
ExecutorService executor =
Executors.newFixedThreadPool(processors);

ArrayList<Future<Integer>> results = new ArrayList<>();

for (int i = 0; i < numberOfTasks; i++) {

    CountPrimesTask oneTask = . . . ;
    Future<Integer> oneResult = executor.submit( oneTask
);
    results.add(oneResult);  // Save the Future
representing the (future) result.
```

```
}
```

The integers that are output by all the subtasks need to be added up to give a final result. The outputs of the subtasks are obtained using the `get()` methods of the `Futures` in the list. Since `get()` blocks until the result is available, the process completes only when all subtasks have finished:

```
int total = 0;
for ( Future<Integer> res : results) {
    try {
        total += res.get();  // Waits for task to
complete!
    }
    catch (Exception e) { // Should not occur in this
program.
    }
}
```

12.3.5 Wait and Notify

Suppose that we wanted to implement our own blocking queue. To do that, 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 can be 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. (I don't know whether they are actually used in Java's standard blocking queue classes, since Java has other ways of solving the wait/notify problem.)

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. 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 can 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, and 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 name of 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 TowersOfHanoiGUI.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. The user can click a "Next Step" button to execute just one step in the solution, which moves a single disk from one pile to another. Clicking "Run" lets the algorithm run automatically on its own; the text on the button changes from "Run" 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, including the buttons:

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 see the new value and 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.

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 are the methods that respond to clicks on the buttons. When the user clicks a button, the corresponding method changes the value of `status` and calls `notify()` to wake up the solution thread:

```
synchronized private void doStopGo() {
    if (status == GO) {  // Animation is running.  Pause
it.
        status = PAUSE;
        nextStepButton.setDisable(false);
        runPauseButton.setText("Run");
    }
    else {  // Animation is paused.  Start it running.
        status = GO;
        nextStepButton.setDisable(true);  // Disabled when
animation is running
        runPauseButton.setText("Pause");
    }
    notify();  // Wake up the thread so it can see the new
status value!
}

synchronized private void doNextStep() {
    status = STEP;
    notify();
}

synchronized private void doRestart() {
    status = RESTART;
    notify();
}
```

These methods are synchronized to allow the calls 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 due to the fact that 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, and the
solution should proceed.
}
```

The `run()` method for the solution thread sets up the initial state of the puzzle and then calls a `solve()` method to solve the puzzle. It runs in an infinite loop so that it can solve the puzzle multiple times. 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. (We used the method of throwing an exception to terminate a recursive algorithm before, in Subsection 12.2.2.)

You can check the full 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!

Section 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. Instead, network programming in Java typically uses **threads**.

12.4.1 The Blocking I/O Problem

As 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. If the buffer fills up, the output method will block until space becomes available in the buffer. 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 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 `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 on port" 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 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 try it on single computer, you can run two copies of the program on that computer, and make a connection between one program window and the other program window, using "localhost" or "127.0.0.1" as the name of the computer. 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.) The

ConnectionHandler handles opening the connection both when the program acts as a server and when it acts as a client. The thread is created when the user clicks either the "Listen" button or the "Connect" button. The "Listen" button makes the thread act as a server, while "Connect" makes it act as a client. To distinguish these two cases, the *ConnectionHandler* class has the two constructors that are shown below. Note that the `postMessage()` method posts a message to the transcript area of the window, where it will be visible to the user:

```
/**
 * 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) {  // For acting as the
"server."
   state = ConnectionState.LISTENING;
   this.port = port;
   postMessage("\nLISTENING ON PORT " + port + "\n");
   try { setDaemon(true); }
   catch (Exception e) {}
   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) {  // For
acting as "client."
   state = ConnectionState.CONNECTING;
   this.remoteHost = remoteHost;
   this.port = port;
   postMessage("\nCONNECTING TO " + remoteHost + " ON PORT
" + port + "\n");
   try { setDaemon(true); }
```

```
   catch (Exception e) {}
   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 6.3.6), 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 to act as a server or as a client when setting up the connection.

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(); // Close
socket and report to user.
         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 and by setting the connection state to `CLOSED`. 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 the GUI 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 by using another thread to transmit outgoing messages.) The `send()` 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 12.5. Also note that the client program,

DateClient.java, which implements a client for the date server, does not need to use threads, since the client only uses one connection. The original client program will work with the new versions of the server.)

As a first attempt, consider DateServerWithThreads.java. This sample program creates a new thread every time a connection request is received, instead of handling the connection itself by calling a subroutine. 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 red italic. Note again that the constructor for the connection thread does very little and in particular cannot block; this is very important since the constructor runs in the main thread:

```
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 a 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; // The connection to the client.
        ConnectionHandler(Socket socket) {
            client = socket;
        }
        public void run() {
                 // (code copied from the original
DateServer program)
            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 (Subsection 12.3.3) named `connectionQueue` for that purpose.
A connection-handling thread takes connections from this queue. Since it is a 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 blocking 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 any 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 that can be used for distributed computing in Java. One general technique that is a a standard part of Java is **RMI** (Remote Method Invocation). RMI enables 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. 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 solves this problem using a "request broker"—a server program running at a known location that 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 must know the location of the broker, and they contact it to find out what services are available and where they are located.

RMI is a complex system that is not very easy to use. I mention it here because they are part of Java's standard network API, but I will not discuss it 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 was used in Section 12.2 and Section 12.3 for MultiprocessingDemo1.java and its variations, namely the computation of a complex image. In this case, however, the program is not a GUI program and the image is not shown on the screen. The computation is one 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 that represents the individual tasks that are performed by the workers. The master divides the overall problem

into a collection of tasks; it distributes those tasks to the workers that will execute the tasks and send the results back to the master; and the master applies the results from all the individual tasks to the overall problem.

To run the demonstration, you must first 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 the master 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`. The worker programs listen 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 172.21.7.101, 172.21.7.102, and 172.21.7.103, then you can run `CLMandelbrotMaster` with the command

```
java  CLMandelbrotMaster  172.21.7.101  172.21.7.102
172.21.7.103
```

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 exits, 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 very 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 `CLMandelbrotWorker` 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 completed, 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 the tasks that must be performed and add them to a
queue
if there are no command line arguments {
      // The master program does all the tasks itself.
   Remove each task from the queue and perform it.
```

```
}
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 send tasks to workers.
   Wait for all threads to terminate.
}
// All tasks are now complete (assuming no error
occurred).
```

The tasks are put into a variable of type *ConcurrentBlockingQueue<CLMandelbrotTask>* named `tasks` (see Subsection 12.3.2.) The communication threads take tasks from this queue 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 in the master program 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()
   if (currentTask != null)
      Encode currentTask into a message and send it to the
worker.
   while (currentTask != null) {
      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.
   }
   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.
}
```

To see how this all translates into Java, check out the *WorkerConnection* nested class in CLMandelbrotMaster.java

Section 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 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 is a fairly complex example, probably the most complex in this book. Understanding it is not essential for a basic understanding of networking.

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 this book. The final example in this section 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 2.6.6, 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 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. Of course, since netgames use JavaFX, you need to use an Eclipse project configured to support JavaFX, as discussed in Section 2.6.

If you work on the command line, you should be in a working directory that includes the *netgame* directory as a subdirectory. You need to add JavaFX options to the `javac` and `java` commands. Let's say that you've defined `jfxc` and `jfx` commands that are equivalent to the `javac` and `java` with JavaFX options included, as discussed in Subsection 2.6.7. Then, to compile all the java files in the package netgame.common, for example, you can use the following command in MacOS or Linux:

```
jfxc netgame/common/*.java
```

For Windows, you should use backslashes instead of forward slashes:

```
jfxc netgame\common\*.java
```

You will need similar commands to compile the source code for the examples in this section, which are defined in other subpackages of `netgame`.

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 *ChatRoomWindow* class, discussed later in this section, is defined in the package netgame.chat, so you would run it with the command

```
jfx netgame.chat.ChatRoomWindow
```

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":

The main things that you need to understand are that: The hub must be running before any clients are started. Clients connect to the hub and can send messages to the hub. The hub processes all messages from clients sequentially, in the order in which they are received. The processing can result in the hub sending messages out to one or more clients. Each client is identified by a unique ID number. This is a *framework* that can be used in a variety of applications, and the messages and processing will be defined by the particular application. Here are some of the details...

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 to the hub and one for receiving messages from the hub. 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 socket 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)
```

When a message from some client arrives at the front of the queue of messages, the message-processing thread removes it from the queue and calls this method. This is the point at which the message from the client is actually processed.

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 a copy of the message to every connected client. This defines the default processing for incoming messages to the hub. 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 default processing might be exactly what is needed by the application. 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 in the *Hub* class that you might want to redefine in a subclass, 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. (The hub has already sent a StatusMessage to every client to inform them about the new player; `playerConnected()` is for any additional actions that a subclass of *Hub* might want to take.) 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 (after the hub sends a *StatusMessage* to the clients). 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. The default value is false. (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. The preferred solution is to use only immutable objects for communication; in that case, no resetting is necessary.)

For more information—and to see how all this is implemented—you should read the source code file Hub.java. With some effort and study, you should be able to understand everything in that file. (However, you only need to understand the public and protected interface of *Hub* and other classes in the netgame framework to write applications based on it.)

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. 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 I/O 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, 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. Note

that ChatRoom uses the *Hub* class itself, not a subclass.

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 run, 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. On the client side in each client, when the message is received from the hub, the `messageReceived()` method of the *ChatClient* object in that client is called. *ChatClient* overrides this method to program it to add the message to the transcript of the *ChatClientWindow*. To summarize: Every message entered by any user is sent to the hub, which just sends out copies of each message that it receives to every client. Each client will see exactly the same stream of messages from the hub.

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 the Hub and is used to send
messages to
 * the Hub 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 bm =
(ForwardedMessage)message;
            addToTranscript("#" + bm.senderID + " SAYS:  "
+ bm.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 );
        Platform.runLater( () -> {
            sendButton.setDisable(true);
            messageInput.setEditable(false);
            messageInput.setDisable(true);
            messageInput.setText("");
        });
        connected = false;
        connection = null;
    }

    /**
     * Posts a message to the transcript when someone
joins 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
```

Except for the constructor, none of the methods in the *ChatClient* class are called by the *ChatRoomWindow* program; they are called from the

connection-handling thread in the client object, which was programmed in Client.java. 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 12.7 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** objects involved: Two objects belonging to a client class, which provide the interface to the two players of the game, and the hub object that manages the connections to the clients. These objects are not even on the same computer, so they certainly can't use the same state 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 for TicTacToe 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. (Instead of sending a complete copy of the state each time the state changes, I might have sent just the change. But that would require some way to encode the changes into messages that can be sent over the network. Since the state is so simple, it seemed easier just to send the entire state as the message in this case.)

Networked TicTacToe is defined in several classes in the package netgame.tictactoe. The class TicTacToeGameState represents the state of a game. It includes a method

```
public void applyMessage(int senderID, Object message)
```

that modifies the state of the game 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 basic *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. When the state of the game changes, a message is sent to each client, and the client's `messageReceived()` method is called to process that message. That method, in turn, calls a `newState()` method in the *TicTacToeWindow* class to update the window. That method is called on the JavaFX application thread using `Platform.runLater()`:

```
protected void messageReceived(Object message) {
    if (message instanceof TicTacToeGameState) {
```

```
        Platform.runLater( () -> newState(
(TicTacToeGameState)message ) );
    }
}
```

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 window 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, and a second window of type *TicTacToeWindow* is opened that immediately connects to the hub. The game will start as soon as a second player connects to the hub. On the other hand, if the user running *Main* chooses to connect to an existing game, then no hub is created. A *TicTacToeWindow* is created, and 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.

(This is the first program that we have seen that uses two different windows. Note that *TicTacToeWindow* is defined as a subclass of *Stage*, the JavaFX class that represents windows. A JavaFX program starts with a "primary stage" that is created by the system and passed as a parameter to the `start()` method. But an application can certainly create additional windows.)

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.

In general outline, the Poker game is similar to the TicTacToe game. There is a Main class that is run by both players. The first player starts a new game;

the second must join that 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 object 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.)

Programming Exercises for Chapter 12

Exercise 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 repeatedly call the `inc()` method in an object of type *Counter*. The object should be a shared global variable. 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).

See the Solution

Exercise 12.2:

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. 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.

See the Solution

Exercise 12.3:

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 results 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.

See the Solution

Exercise 12.4:

In previous exercise, you used a thread pool and a queue of tasks to find the integer in the range 1 to 100000 that has the largest number of divisors. Subsection 12.3.4 discusses a higher-level approach that uses an *ExecutorService*. Write one more program to solve the problem, this time using an *ExecutorService* and *Futures*. The program should still break up the computation into a fairly large number of fairly small tasks, and it should still print out the largest number of divisors and the integer that has that number of divisors.

(There is yet another way to solve the same problem: the stream API from Section 10.6. My on-line solution of this exercise also discusses how to use streams to solve the problem.)

See the Solution

Exercise 12.5:

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.

See the Solution

Exercise 12.6:

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 π/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 π/4. If we choose a point in the square at random, the probability that it is inside the circle is π/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 π/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 point 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 *Labels*

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 12.3.5.)

Here is a picture of the program after it has run many 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);
Platform.runLater( () -> {
    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.)

See the Solution

Exercise 12.7:

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 established. It should throw an *IOException* if some error occurs during the setup process. Note that any messages that are sent by the hub as part of the handshake must 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. (Just imitate the code that asks for the host name.) You will have to decide what to do if two users want to use the same name.

For the second improvement, personal messages, I suggest writing a new *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 of 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 by the hub 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. Don't forget that *PrivateMessage* needs to be declared to implement *Serializable*.

If you attempt this exercise, you are likely to find it quite challenging.

See the Solution

Quiz on Chapter 12

Question 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 creates and starts a thread belonging to that class.

Question 2:

Suppose that `thrd` is an object of type *Thread*. Explain the difference between calling `thrd.start()` and calling `thrd.run()`.

Question 3:

What is a *race condition*?

Question 4:

How does synchronization prevent race conditions, and what does it mean to say that synchronization only provides *mutual* exclusion?

Question 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?

Question 6:

What is an *ArrayBlockingQueue* and how does it solve the producer/consumer problem?

Question 7:

What is a *thread pool*?

Question 8:

Network server programs are often *multithreaded.* Explain what this means and why it is true.

Question 9:

Why does a multithreaded network server program often use many times more threads than the number of available processors?

Question 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.)

See the Answers

Chapter 13

GUI Programming Continued

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 JavaFX 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 continues the study of JavaFX, but still does not cover it fully. Some advanced topics, such as support for video and three-dimensional graphics, are not even mentioned. Full coverage of JavaFX would require at least another complete book. However, this chapter should deepen your understanding of JavaFX and GUI programming.

Contents of Chapter 13:

- Section 1: Properties and Bindings
- Section 2: Fancier Graphics
- Section 3: Complex Components and MVC
- Section 4: Mostly Windows and Dialogs
- Section 5: Finishing Touches
- Programming Exercises
- Quiz on This Chapter

Section 13.1

Properties and Bindings

WE HAVE SEEN THAT GUI PROGRAMMING makes heavy use of events, including low-level mouse and keyboard events and higher-level events such as those that are generated when the user makes a menu selection or adjusts the value of a slider. In Subsection 6.4.5, we saw that events from a slider actually come from an "observable property" of the slider. That is, in order to respond to changes in the value of a *Slider*, `sldr`, you need to register a listener with the `valueProperty` of the slider:

```
sldr.valueProperty().addListener( . . . );
```

When the value of the slider changes, its `valueProperty` emits an event that allows your event-handling code to respond to the change.

In fact, the value property of a slider is an object of type *DoubleProperty*. A *DoubleProperty* has several aspects. First of all, it wraps a value of type double, and it has methods `get()` and `set()` for retrieving and changing that value. Second, it is an "observable value," which means that it emits events when the double value is changed. Third, it is a "bindable property," an aspect that is possibly unique to JavaFX. A bindable property can be bound to another property of the same type. Properties that are bound in this way are forced to have the same value. Bindable properties are used throughout the JavaFX API. This section explores how and why they are used.

The classes discussed in this section are defined in package `javafx.beans` and its subpackages. However, you will rarely need to import classes from those packages into your JavaFX programs, since you will mostly be using objects that already exist.

13.1.1 Observable Values

Many instance variables in JavaFX objects are **observable values**, meaning that they emit events. Many of the observables are actually **observable properties** in the sense that they are bindable. The width and height of a canvas are observables of type *DoubleProperty*. The text in a text field or label is an observable of type *StringProperty*. The list of children of a *Pane* is an observable of type `ObservableList<Node>`. A checkbox has observable property of type *BooleanProperty* that says whether the box is currently checked. The color of the text in a label is an observable of type `ObjectProperty<Paint>`.

An observable value emits two kinds of events. It emits a change event whenever the value changes. A handler for the event must implement the parameterized functional interface `ChangeListener<T>`, which defines a method `changed(target,oldValue,newValue)`. The first parameter of `change()` is the observable whose value has changed; the second parameter is the previous value of that observable; and the third parameter is the new value. Suppose, for example, we want the text on a label to display the current value of a slider. When the slider value changes, we need to change the text on the label to match. One way to do that is to register a change listener with the property that records the value of the slider:

```
slider.valueProperty().addListener(
            (t,oldVal,newVal) -> label.setText("Slider
Value: " + newVal) );
```

However, we will see below that there is a better way to accomplish the same thing.

The second type of event emitted by an observable value is an invalidation event, which is emitted when the current value becomes invalid for some reason. An invalid value will never be seen by another object; any attempt to read the value will cause the value to be recomputed and become valid again. An invalidation event is a notice that the value needs to be recomputed, but the value won't actually be recomputed until it is needed. This is something called "lazy evaluation" of the value: The work of recomputing the value is not done until the new value is actually needed for something. Lazy evaluation can be more efficient than recomputing the value every time it is invalidated. Let's say a dozen things happen that invalidate

the value before the value is needed—with lazy evaluation, the value will only be recomputed once, when it is needed, rather than a dozen times. More important for us, the observable would only emit one invalidation event, not a dozen. This avoids multiple redundant calls to invalidation listeners. In most cases in JavaFX, invalidation listeners should be used in preference to change listeners.

An *InvalidationListener* defines the single method `invalidated(obv)`, where the parameter is the observable that has been invalidated. Consider this code, where `sayHello` is a *CheckBox*:

```
sayHello.selectedProperty().addListener( e -> {
    if (sayHello.isSelected())
        label.settext("Hello");
    else
        label.setText("Goodbye");
});
```

In this example, an *InvalidationListener* is registered as a listener. You could accomplish almost the same thing using `sayHello.setOnAction()`. However, an *ActionListener* would be called only when the **user** changes the selected state of the checkbox, while the invalidation listener is called whenever the value changes, including changes made by calling `sayHello.setSelected()`. Note that *ChangeListeners* are also registered using a method named `addListener()`. The compiler can tell the difference, even when the listener is given by a lambda expression, because a lambda expression for a *ChangeListener* has three parameters, while a lambda expression for an *InvalidationListener* has only one parameter.

You might have asked yourself what would happen if the `selectedProperty` of `sayHello` was merely invalidated rather than changed. Would `sayHello.isSelected()` return the current invalid value or the new value? In fact, calling `sayHello.isSelected()` would force the computation of the new value, and it would return the new value. An invalid value is never seen because any attempt to read it forces the new value to be computed.

The general pattern in JavaFX is that an observable property of an object is accessed by calling an instance method whose name ends with "Property". For example, the value property of a slider is given by

`slider.valueProperty()`, and the text property of a label is given by `label.textProperty()`. Some JavaFX objects have observable values that are not properties. An observable property is bindable, as we will see in the next section. A plain observable value is not bindable, but it does emit change events and invalidation events, and you can register listeners if you want to respond to changes in the value.

13.1.2 Bindable Properties

Many JavaFX observable properties can be bound to another property of the same type. (The exceptions are "read-only properties"; you can't bind a read-only property to another property, although you **can** bind a property **to** a read-only property.) As a basic example, suppose that we want the text in a label to always be the same as the text in a text field. That can be implemented simply by binding the `textProperty` of the label to the `textProperty` of the text field:

```
Label message = new Label("Never Seen");
TextField input = new TextField("Type Here!");
message.textProperty().bind( input.textProperty() );
```

The `bind()` method forces the value of `message.textProperty()` to be the same as the value of `input.textProperty()`. As soon as `bind()` is executed, the text from the text field is copied into the label, so that the initial text in the label—"Never Seen" in the example—is never seen by the user. As the program runs, any change to the text in the text field will automatically be applied to the label as well, whether that change is due to the user typing in the text field or due to `input.setText()` being called. Bindings are implemented internally using events and listeners, but the point is that you don't have to set up the listeners and respond to the events yourself—everything is set up by the `bind()` method.

When `bind()` is used to bind a property to another property, it becomes illegal to change the value of the bound property directly. In the example, any call to `message.setText()` would throw an exception. And of course, a property can only be bound to one other property at a time. The `unbind()` method, which takes no parameter, can be used to remove a binding:

```
message.textProperty().unbind();
```

The sample program BoundPropertyDemo.java contains several examples of bound properties. In particular, the text property of a large label is bound to the text property of a text field, as in the preceding example, so that typing in the text field also changes the text on the label. Here is a screenshot:

The label at the lower right of this window provides another example of binding. The label displays the value of the slider, and the text on the label will change as the user adjusts the slider value. As noted at the beginning of this section, one way to implement this interaction would be to register a listener with the slider's `valueProperty`. Here, however, it's done with a binding. Now, the `textProprety` of the label is a *StringProperty* while the `valueProperty` of the slider is a *DoubleProperty*, so it's not possible to directly bind the two values. A binding only works for properties of the same type. However, a *DoubleProperty* has a method, `asString()` that converts the property into a string property. That is, if `slider` is a *Slider*, then

```
slider.valueProperty().asString()
```

is a string property representing the double value of the slider as a string. The text property of a label can be bound to that string property. In fact, `asString()` can take a format string (like the ones used with `System.out.printf`) as an optional parameter that is used to format the double value. In the program, the label is `sliderVal`, and its text property is bound by saying:

```
sliderVal.textProperty().bind(
                    slider.valueProperty().asString("Slider
Value: %1.2f") );
```

(Actually, `slider.valueProperty().asString` is of type *StringBinding* rather than *StringProperty*, but the distinction is not important here, and I will ignore it.)

Property objects have many methods for converting properties of one type into properties of another type, as well as other operations. For another example, a *DoubleProperty* has a method `lessThan(number)` that returns a boolean property whose value is `true` when the value of the *DoubleProperty* is less than `number`. For example, a *Button*, `btn` has a *BooleanProperty*, `btn.disableProperty()` that tells whether the button is disabled. If we want the button to be disabled when the value on a slider is less than 20, we can do that by binding the button's disable property as follows:

```
btn.disableProperty().bind(
slider.valueProperty().lessThan(20) );
```

There are similar methods `greaterThan()`, `lessThanOrEqual()`, `isNotEqualTo()`, and so on. There are also methods for doing math. For example,

```
slider.valueProperty().multiply(2)
```

is a double property whose value is 2 times the value of the slider.

The class *When*, from package `javafx.beans.binding`, can be used to apply something like the ternary operator, "?:" (see Subsection 2.5.5), to boolean properties, with a rather strange syntax. If `boolProp` is a boolean property, and if `trueVal` and `falseVal` are any values that both have the same type, then

```
new When(boolProp).then(trueVal).otherwise(falseVal)
```

represents a property whose type is the same as the type of `trueVal` and `falseVal`. The value of this property is `trueVal` if `boolProp` has the value `true`, and the value is `falseVal` if `boolProp` has the value `false`.

In an earlier example, we used a listener to set the text in a label to "Hello" or "Goodbye" depending on whether or not a checkbox, `sayHello`, is

checked. Here's how to do the same thing with a property binding:

```
label.textProperty().bind(
     new
When(sayHello.selectedProperty()).then("Hello").otherwise(
"Goodbye")
);
```

The parameter in `new When(sayHello.selectedProperty())` is a boolean property. Since "Hello" and "Goodbye" are values of type *String*, the property that results from the complete expression is a string property, which matches the type of `label.textProperty()`.

The same sort of thing is done in BoundPropertyDemo.java to control the background color of the big label. The background is controlled by a checkbox using binding and an object of type *When*. See the well-commented source code for details.

A straightforward, but useful, application of property binding can be found in the sample program CanvasResizeDemo.java. The program shows fifty small red disks that bounce around in a window. The disks are drawn to a *Canvas* that fills the window, and the disks bounce off the edges of the canvas. In previous examples that used a canvas, the window was made non-resizable because canvasses do not automatically change size. However, it is possible for a program to change the size of a canvas by setting the canvas's height and width properties. That can be done by calling `canvas.setWidth(w)` and `canvas.setHeight(h)`. However, the height and width are bindable properties of type *DoubleProperty*, so another way to set the canvas size is to bind these properties to an appropriate source.

In the program, the canvas is contained in a *Pane* that serves as the root of the scene graph and fills the window. When the size of the window is changed by the user, the size of the *Pane* is automatically set to match. If we want the canvas size to track the size of the pane, it is only necessary to bind the width property of the canvas to the width property of the pane, and to bind the height property of the canvas to the height property of the pane. That is done in the program with two lines of code:

```
canvas.widthProperty().bind( root.widthProperty() );
canvas.heightProperty().bind( root.heightProperty() );
```

If you run the program and increase the size of the window, you can see that the canvas has also increased in size, because the bouncing red disks will spread out to fill the larger space. Similarly, if you decrease the size, the disks will be trapped in the smaller space.

In this program, the canvas is being continually redrawn, so no special action needs to be taken to redraw it when the canvas size is changed. In some programs, it might be necessary to redraw the contents of the canvas when its size changes. One way to make that happen is to add listeners to the width and height properties of the canvas. The listeners can redraw the canvas in response to changes in the width or height.

13.1.3 Bidirectional Bindings

Bindings created using the `bind()` method are one-way bindings: They only work in one direction. One-way bindings are not always appropriate. Suppose for example that `cb1` and `cb2` are of type *CheckBox*, and that we would like the two checkboxes to always be in the same state. That can't be done with a one-way binding. If we do

```
cb2.selectedProperty().bind( cb1.selectedProperty() );
```

that will make `cb2` match its state to `cb1`. But changing the selected state of `cb2` will not change the state of `cb1`; instead, it will cause an exception, since it is illegal to try to change the value of a property that has been bound using `bind()`. In fact, if the user just clicks `cb2`, there will be an exception as the checkbox tries to change state.

The solution that we need here is something called a **bidirectional binding**. When two properties are bound bidirectionally, then the value of either property can be changed, and the other property will automatically be changed to the same value. Bidirectional bindings are set up with the method `bindBidirectional()`. For our two checkboxes, we can say:

```
cb2.selectedProperty().bindBidirectional(
cb1.selectedProperty() );
```

Now, the user can click either checkbox, and the other will also change state. This would probably not be useful for two checkboxes, but it can be useful to synchronize the state of a *CheckBox* in a window with the state of a *CheckMenuItem* in a menu. The user would have a choice of using either interface element. Similar redundant interface elements are common in menus and toolbars.

The sample program BoundPropertyDemo.java does something similar with *RadioButtons* and *RadioMenuItems*. The color of the label in the program can be controlled using either a "Color" menu or a set of radio buttons. The state of each *RadioButton* is bidirectionally bound to the state of a corresponding *RadioMenuItem*. It might be worth looking at how it's done in detail:

```
Menu colorMenu = new Menu("Color");

Color[] colors = { Color.BLACK, Color.RED, Color.GREEN,
Color.BLUE };
String[] colorNames = { "Black", "Red", "Green", "Blue" };

ToggleGroup colorGroup = new ToggleGroup();

for (int i = 0; i < colors.length; i++) {

        // Make a menu item and corresponding radio
button.
    RadioButton button = new RadioButton(colorNames[i]);
    RadioMenuItem menuItem = new
RadioMenuItem(colorNames[i]);

    button.selectedProperty().bindBidirectional(
menuItem.selectedProperty() );

    menuItem.setToggleGroup(colorGroup);

        // Note how UserData is used to store the color
object
        // associated with the menu item, for later use.
    menuItem.setUserData(colors[i]);

    right.getChildren().add(button);    // add button to a
```

```
container
    colorMenu.getItems().add(menuItem); // add menu item
to a menu
    if (i == 0)
        menuItem.setSelected(true);
}

colorGroup.selectedToggleProperty().addListener( e -> {
            // Listen for changes to the
selectedToggleProperty
            // of the ToggleGroup, so that the color of
the
            // label can be set to match the selected menu
item.
    Toggle t = colorGroup.getSelectedToggle();
    if (t != null) {
            // t is the selected RadioMenuItem.  Get the
color
            // from its UserData, and use it to set the
color
            // of the text.  The value of the
selectedToggleProperty()
            // can momentarily be null as one toggle is
unselected
            // and another is selected.
        Color c = (Color)t.getUserData();
        message.setTextFill(c);
    }
});
```

Note that the menu items are added to a *ToggleGroup* (see Subsection 6.4.3), but the buttons are not. Suppose that the user clicks on a currently unselected radio button. The state of the button changes to "selected." Because of the bidirectional binding, the state of the corresponding radio menu item must also be changed to "selected." Before that can happen, however, the *ToggleGroup* will change the state of the currently selected radio menu item to "unselected," which will in turn cause the state of its associated radio button to change to "unselected". In the end, the states of two radio buttons and two radio menu items are changed.

(Although it is not relevant to property binding, you should also check out how this code uses the `userData` property of the radio buttons. Every scene graph node has user data of type *Object*, which is not used by the system. The `userData` property of a node can be a convenient place for a programmer to

store data of any type that the programmer wants to associate with the node. In this case, the user data for a radio menu item is a value of type *Color*, and that color value is used when the menu item is selected.)

I hope that the examples in this section have convinced you that property binding can be an effective way to program interactions between objects in a JavaFX program. This style of programming might look strange at first, but it can be easier and clearer than working directly with events and listeners.

Section 13.2

Fancier Graphics

WE HAVE SEEN MANY EXAMPLES of using a *GraphicsContext* to draw on a *Canvas*. But *GraphicsContext* has many features besides those that have already been covered. This section continues our study of canvas graphics. We begin by looking at two methods that can be useful for managing the state of a graphics context.

A *GraphicsContext*, `g`, has various properties, such as fill color and line width, that affect any drawing done with `g`. It's important to remember that a canvas has only one graphics context, and any changes made to a property of the graphics context will carry over to all future drawing, until the value of the property is changed again. In particular, changes are **not** restricted to the subroutine in which they are made. It often happens that a programmer wants to change the value of some properties temporarily and then restore them to their previous values. The graphics API includes methods `g.save()` and `g.restore()` that makes this easy. When `g.save()` is executed, the current state of the graphics context, `g`, is stored. The state includes almost all of the properties that affect drawing. In fact, the graphics context keeps a **stack** of states (see Subsection 9.3.1), and `g.save()` pushes the current state onto that stack. A call to `g.restore()` pops the top state from the stack, and sets the values of all properties to match their saved values.

Because they use a stack, it is possible to call `save()` several times before calling `restore()`. Every call to `save()` should be matched, eventually, by a call to `restore()`. (However, unmatched calls to `restore()` will not produce an error; the extra calls are simply ignored.) An easy way to make sure that changes made to a graphics context in one subroutine do not carry over to future subroutine calls is to call `save()` at the beginning of the subroutine and `restore()` at the end.

`Save` and `restore` are particularly useful when working with transforms, which are covered later in this section.

13.2.1 Fancier Strokes

We have seen how to stroke lines and curves and even the outlines of text characters, and how to set the color and line width that are used for the stroke. Strokes have other properties that affect their appearance. It is possible to draw strokes that are dotted or dashed instead of solid. And you can control what happens at the endpoints of a stroke, and at the "join" between two lines or curves that meet at a point, such as two sides of a rectangle meeting at one of the corners of the rectangle. A graphics context has properties to control all of these features. You need to be working with fairly wide lines for the endpoint and join properties to have any visual effect. This illustration shows the available options:

The endpoints of the three line segments on the left show the three possible styles of line "cap." The stroke is shown in black, and the geometric line is indicated by a yellow line down the center of the stroke. In the `BUTT` cap style, the stroke is simply cut off at the end of the geometric line. For the `ROUND` cap, a disk is added at the endpoint, with a diameter equal to the line width. For the `SQUARE` cap, a square is added instead of a disk. The round or square style is what you would get if you draw a stroke with a physical pen that has a round or square tip.

In a graphics context, `g`, the cap style for strokes is set by calling `g.setLineCap(cap)`. The parameter is of type *StrokeLineCap*, an enumerated type in package `javafx.scene.shape` whose possible values are `StrokeLineCap.BUTT`, `StrokeLineCap.ROUND`, and `StrokeLineCap.SQUARE`. The default style is `SQUARE`.

Line joins are similar. The appearance of a vertex where two lines or curves meet is set by calling `g.setLineJoin(join)`. The parameter is of type *StrokeLineJoin*, and the possible values are `StrokeLineJoin.MITER`, `StrokeLineJoin.ROUND`, and `StrokeLineJoin.BEVEL`. The three styles are shown in the middle section of the above illustration. The default join style is `MITER`. In the miter style, the two line segments are continued to make a sharp point. In the other two styles, the corner is cut off; for bevel, it is cut off by a line segment, and for round it is cut off by an arc of a circle. I have found that round joins can look better if you draw a wide curve as a sequence of short line segments.

Dotted and dashed strokes can be made by setting a dash pattern, using the method `g.setLineDashes()`. The parameters to this method specify the lengths of the dashes and of the gaps between them:

```
g.setLineDashes( dash1, gap1, dash2, gap2, . . . );
```

The parameters are of type double (and could also be given as a single array of type `double[]`). If a stroke is drawn when this dash pattern is in effect, the curve consists of a line or curve of length `dash1`, followed by a gap of length `gap1`, followed by a line or curve of length `dash2`, and so on. The pattern of dashes and gaps will be repeated as often as necessary to cover the full length of the stroke.

For example, `g.setLineDashes(5,5)` draws a stroke as series of segments of length 5 followed by gaps of length 5, while `g.setLineDashes(10,2)` produces a series of long segments separated by short gaps. A pattern of dashes and dots could be specified, for example, as `g.setLineDashes(10,2,2,2)`. The default dash pattern, of course, is a solid line with no dots or dashes.

The sample program StrokeDemo.java lets you draw lines and rectangles with a variety of line styles. See the source code for details.

13.2.2 Fancier Paints

So now we can draw fancier strokes. But all of our individual drawing operations have been 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 abstract class in package `javafx.scene.paint`. The *Color* class is just one concrete subclass of *Paint*. Any object of type *Paint* can be used as the parameter to `g.setFill()` and `g.setStroke()`. When a *Color* is used as the paint, 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. JavaFX has several classes that define paint with this property: *ImagePattern* and two kinds of gradient paint. In an image pattern, the pixel colors come from an image, which is repeated, if necessary, like a wallpaper pattern to cover the entire xy-plane. In a **gradient**, the color that is applied to pixels changes gradually from one color to another color as you move from point to point. Java has two gradient paint classes, *LinearGradient* and *RadialGradient*.

It will be helpful to look at some examples. The following illustration shows a polygon filled with two different paints. The polygon on the left uses a *LinearGradient* while the one on the right uses an *ImagePattern*. 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 stroking shapes as well as for filling them. 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.

To create an *ImagePattern*, you first need an image. The *Image* class was introduced in Subsection 6.2.3. As we saw then, an *Image* object can easily

be constructed from an image resource file, which is considered to be part of the program that uses it. Given an *Image*, `pict`, an `ImagePattern` can be created with a constructor of the form

```
patternPaint = new ImagePattern( pict, x, y, width,
height, proportional );
```

The parameters `x`, `y`, `width`, and `height` are double values that specify the position and size of the image on the canvas. One copy of the image is placed with its top left corner at `(x,y)`, and is scaled to the given width and height. The image is then repeated horizontally and vertically to fill the canvas (but you only see the part of the pattern that is within the shape to which the paint is applied).

The last parameter to the constructor is a boolean that specifies how `x`, `y`, `width` and `height` are interpreted. If `proportional` is false, the width and height are measured in the usual coordinate system on the canvas. If `proportional` is true, then `width` and `height` are measured in multiples of the size of the shape to which the paint is applied, and `(x,y)` is `(0,0)` at the upper left corner of the shape (more precisely, of the rectangle that just contains the shape). For example,

```
patternPaint = new ImagePattern( pict, 0, 0, 1, 1, true );
```

creates an image paint where one copy of the image just covers the shape. If the paint is applied to several shapes of different sizes, the paint will be scaled differently for each shape. If you want the shape to contain four copies of the shape horizontally and two copies vertically, you would use

```
patternPaint = new ImagePattern( pict, 0, 0, 0.25, 0.5,
true );
```

A linear gradient is defined by specifying a line segment and the color of several points along the line segment. The points and their associated colors are called **color stops**. Colors are interpolated between the color stops to give a color to each point on the line. The colors are then extended perpendicularly to the line segment to give an infinite colored strip. You also have to specify what happens outside that strip of color. In JavaFX, this is

done by specifying a "cycle method." The possibilities are `CycleMethod.REPEAT`, meaning that the color strip is repeated to cover the entire plane; `CycleMethod.MIRROR`, meaning that the color strip is repeated, but every other copy is reversed so that the colors match up at the boundaries; and `CycleMethod.NO_REPEAT`, meaning that the color along each boundary is extended infinitely. Here are pictures of three red/blue/yellow gradients using the same original line segment and color stops. The picture on the left uses the `MIRROR` cycle method, the one in the middle uses `REPEAT`, and the one on the right uses `NO_REPEAT`. The basic line segments for the gradients are shown, and the positions of the three color stops on that line segment are marked:

The constructor for a linear gradient paint takes the form

```
linearGradient = new LinearGradient( x1,y1, x2,y2,
proportional, cycleMethod,
                                                    stop1,
stop2, . . . );
```

The first four parameters are double values that specify the endpoints of the line segment: `(x1,y1)` and `(x2,y2)`. The fifth parameter, `proportional`, is a boolean; if false, the endpoints are specified in the usual coordinate system of the canvas; if true, they are specified in a coordinate system in which (0,0) is at the upper left corner of the shape to which the paint is applied, and (1,1) is at the lower right corner. The sixth parameter, `cycleMethod`, is one of the constants `CycleMethod.REPEAT`, `CycleMethod.MIRROR`, or `CycleMethod.NO_REPEAT`. The remaining parameters are the color stops for the gradient. A color stop is given as an object of type *Stop*. The constructor takes a double and a *Color*. The first parameter specifies the position of the stop along the basic line segment as a fraction of the distance from the first endpoint to the second endpoint. In general, the first color stop should have

position 0, and the last one should have position 1, and the position value for each color stop should be greater than the position of the preceding color stop. As an example, the gradient paint for the first gradient in the above illustration was constructed as

```
grad = new LinearGradient( 120,120, 200,180, false,
CycleMethod.MIRROR,
                                    new Stop( 0,
Color.color(1, 0.3, 0.3) ),
                                    new Stop( 0.5,
Color.color(0.3, 0.3, 1) ),
                                    new Stop( 1,
Color.color(1, 1, 0.3)   )  );
```

For a linear gradient, the color is constant along certain lines. For a radial gradient, color is constant along certain circles. For a basic radial gradient, color stops are specified along the radius of a circle, and color is constant along circles that have the same center as that circle. Colors outside the circle are determined based on the cycle method. Things are a little more complicated than that because a radial gradient can have a "focal point" inside the circle. For a basic radial gradient, the focal point is the center of the circle but the focal point can be any point inside the circle. The color of the gradient at the focal point is given by the color stop at position 0. The color along the circle is given by the color stop at position 1. It's easiest to see this in an illustration. The gradients in this picture are the same, except for the focal point. In the top row, the circle and the focal point are marked. Note that in all cases, the gradient is red at the focal point and yellow along the circle:

The constructor for a radial gradient has a lot of parameters...

```
radialGradient = new RadialGradient(
focalAngle,focalDistance,
                              centerX,centerY,radius,
                              proportional, cycleMethod,
                              stop1, stop2, . . . );
```

The first two parameters specify the location of the focus. The focal distance is the distance of the focus from the center of the circle, given as a fraction of the radius; it must be strictly less than 1. The focal angle is the direction in which the focus is moved away from the center. For a basic radial gradient, the focal distance is 0, and the focal angle is irrelevant. The next three parameters specify the center and radius of the circle, and the remaining parameters are similar to the same parameters for linear gradients.

13.2.3 Transforms

In the standard drawing coordinates on a canvas, the upper left corner of the component has coordinates `(0,0)`, 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. However, you are not restricted to using these coordinates. In fact, you can set up a graphics context to use other coordinate

systems, with different units of length and different coordinate axes. 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** (or "transformations"). 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. Less common is a **shear** transform, which "tilts" an image. This illustration shows an original picture and several transformed copies of the picture:

Notice that everything in the image, including the text, is affected by a transform.

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, and transforms are a major topic in a course in computer graphics. I will limit myself here to

discussing a few simple cases, just to give you an idea of what transforms can do.

The current overall transform is a property of the graphics context. It is part of the state that is saved by the `save()` method and restored by the `restore()` method. It is especially important to use `save()` and `restore()` when working with transforms, to prevent the effect of a transform from carrying over from one subroutine call to another.

You should also remember that, like other properties of a graphics context, a transform affects things that are drawn **after** the transform is applied to the graphics context. It does not transform the picture that has already been drawn.

Suppose that `g` is of type *GraphicsContext*. A translation transform can be applied to `g` by calling `g.translate(x,y)`, where `x` and `y` are of type double. Mathematically, the effect is to automatically add `(x,y)` to coordinates in subsequent drawing operations. For example, if you use coordinates `(0,0)` after applying the translation, you are actually referring to the point that had coordinates `(x,y)` **in the usual coordinate system**. All other coordinate pairs are moved by the same amount. For example the two statements

```
g.translate(x,y);
g.strokeLine( 0, 0, 100, 200 );
```

draws the same line as the single statement

```
g.strokeLine( x, y, 100+x, 200+y );
```

In the second code segment, you are just doing the same translation "by hand." 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 `g.translate(x,y)`, any objects that you draw are displaced `x` units horizontally and `y` units vertically.

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 before drawing anything:

```
g.translate( canvas.getWidth()/2, canvas.getHeight()/2 );
```

To apply a scale transform to `g`, use `g.scale(sx,sy)`, where the parameters specify the scaling factor in the x-direction and the y-direction. After this command, x-coordinates are multiplied by `sx`, and y-coordinates are multiplied by `sy`. The effect of scaling is to make objects bigger or smaller. Scale factors greater than 1 will magnify sizes of shapes, while scale factors less than 1 will shrink the shapes. In many cases, the two scale factors are the same; this is called "uniform scaling."

The center of scaling is `(0,0)`. That is, the point `(0,0)` is unaffected by the scaling, and other points move toward or away from `(0,0)`. If an object is not located at `(0,0)`, then the effect of scaling is not just to change its size but also to move it farther away from `(0,0)` (for scaling factors greater than 1) or closer to `(0,0)` (for scaling factors less than 1).

It is possible to use a negative scaling factor, which results in a reflection. For example, after calling `g.scale(-1,1)`, objects will be reflected horizontally through the line `x=0`.

The third type of basic transform is rotation. The command `g.rotate(r)` rotates all subsequently drawn objects through an angle of `r` about the point `(0,0)`. Angles are measured in degrees. Positive angles are clockwise rotations, while negative angles are counterclockwise (unless you have applied a negative scale factor, which reverses the direction of rotation).

Shearing is not considered a basic transform, since it can be done (with some difficulty) by a series of rotations and scalings. The effect of a horizontal shear is to shift horizontal lines to the left or right by an amount that is proportional to the distance from the x-axis. That is, the point `(x,y)` is transformed to `(x+a*y,y)`, where `a` is the amount of shear. JavaFX does not have a method that applies a shear transform, but it does have a way to apply an arbitrary transform. For those who know some linear algebra, transforms are represented as matrices, and it is possible to specify a transform directly by giving the numbers that go in the matrix. A transform in JavaFX is represented by an object of type *Affine*, and the method `g.transform(t)` applies the *Affine* transform `t` to the graphics context. I don't want to go into

the math here, but to do a horizontal shear with shear amount equal to `a`, you can use

```
g.transform( new Affine(1, a, 0, 0, 1, 0) );
```

Sometimes you will need to apply several transforms to get the effect you want. Suppose, for example, that you would like to show the string "hello world" tilted at a rising 30-degree angle and with its basepoint at `(x,y)`. This **won't** do it:

```
g.rotate(-30);
g.fillText("hello world", x, y);
```

The problem is that the rotation applies to the point `(x,y)` as well as to the text. After the rotation, the basepoint is no longer at `(x,y)`. What you need to do is make a rotated string with its basepoint at `(0,0)`, and then translate by `(x,y)`, which will move the basepoint from `(0,0)` to `(x,y)`. That can be done as follows:

```
g.translate(x,y);
g.rotate(-30);
g.fillText("hello world", 0, 0);
```

The important thing to note is the order of the transforms. The translation applies to everything that comes after the `translate` command. What comes after it is some code that draws a rotated string with its basepoint at `(0,0)`. That rotated string will be translated by `(x,y)`. In effect, the string is first rotated, then translated. The rule for multiple transforms is that **transforms are applied to objects in the opposite of the order in which the transform commands occur in the code.**

The sample program TransformDemo.java can apply various transformations to a picture. The user controls the amount of scaling, horizontal shear, rotation and translation. Running the program might help you understand transforms, and you can read the source code to see how to code the transforms. Note that the transforms in this program are applied to the objects in the order scale-shear-rotate-translate. If you look in the code, you will see that the transform methods are called in the opposite order, translate-rotate-shear-scale. There is also an additional translation that moves the origin to

the center of the canvas, so that the center of scaling, rotation, and shearing is at the center of the canvas.

13.2.4 Stacked Canvasses

The final example for this section is another simple paint program, ToolPaint.java. The paint programs in Chapter 6 could only draw curves. In the new program, the user can select a "tool" for drawing. In addition to a curve tool there are tools for drawing five kinds of shape: straight lines, rectangles, ovals, filled rectangles, and filled ovals.

When the user draws with one of the five shape tools, the user drags the mouse and a shape is drawn between the starting point of the drag gesture and the current position of the mouse. Each time the mouse moves, the previous shape is deleted and a new shape is drawn. The new shape doesn't actually become part of the picture until the user releases the mouse button. For the line tool, for example, the effect is that the user sees a line that stretches from the starting point to the current mouse position, and that line moves along with the mouse. I urge you to try the program to see the effect!

The difficulty for the programmer is that parts of the drawing are continually being covered and uncovered as the user moves the mouse around. When part of the current drawing is covered and then uncovered, the drawing must still be there! This means that while the user is dragging the mouse, the program can't simply draw the new shape on the same canvas that contains the current picture, since doing so would obliterate whatever was in that part of the picture.

The solution in JavaFX is to use **two** canvases, one stacked on top of the other. The bottom canvas contains the actual drawing. The top canvas is used to implement the shape tools. Ordinarily, the top canvas is completely transparent. That is, it is filled with a color that has alpha component zero. When a shape tool is being used, the shapes that are drawn as the user drags the mouse are drawn to the top canvas. The bottom canvas is not affected, but part of it is hidden behind the shape that was drawn in the top canvas. Each time the mouse moves, the top canvas is cleared, and a new shape is drawn

to the top canvas. When the user releases the mouse at the end of the drag, the top canvas is cleared, and the shape is drawn to the bottom canvas, where it becomes part of the actual drawing. In the end, the top canvas is once again completely transparent, and the drawing in the bottom canvas is fully visible. Note that a canvas can be cleared to full transparency by calling

```
g.clearRect( 0, 0, canvas.getWidth(), canvas.getHeight()
);
```

where `g` is a *GraphicsContext* for the canvas. (Canvasses are also fully transparent when they are first created.)

To stack one canvas on top of another, you can use a *StackPane*. A *StackPane* will simply stack up its child nodes one on top of the other, in the order in which they were added to the pane. The two canvasses in the program were set up like this:

```
canvas = new Canvas(width,height); // main drawing canvas
canvasGraphics = canvas.getGraphicsContext2D();
canvasGraphics.setFill(backgroundColor);
canvasGraphics.fillRect(0,0,width,height);

overlay = new Canvas(width,height); // transparent top
canvas
overlayGraphics = overlay.getGraphicsContext2D();
overlay.setOnMousePressed( e -> mousePressed(e) );
overlay.setOnMouseDragged( e -> mouseDragged(e) );
overlay.setOnMouseReleased( e -> mouseReleased(e) );

StackPane canvasHolder = new StackPane(canvas,overlay);
```

Note that the mouse event handlers were added to the top canvas, since the top canvas covers the bottom canvas. When the user clicks on the drawing, it is actually the top canvas that is being clicked.

The curve tool, by the way, does not use the top canvas. Curves are drawn directly to the bottom canvas. Since no part of the curve will ever be deleted after it is drawn, there is no need to put a temporary copy in the top canvas.

13.2.5 Pixel Manipulation

ToolPaint.java has two more tools: "Erase" and "Smudge." (Both of these tools work directly on the bottom canvas.) The Erase tool does what it says: as the user drags the mouse, a small square around the mouse location is filled with the background color, erasing part of the drawing at that location. The Smudge tool is more interesting. Dragging with the smudge tool smears the color under the tool, as if you are dragging your finger through wet paint. In this picture from the program, the smudge tool was dragged around on the center of a red rectangle:

There is no built-in subroutine in JavaFX for smudging a picture. This is something that requires direct manipulation of the colors of individual pixels. Here is the basic idea: To implement the smudge tool, the program uses three 9-by-9 two-dimensional arrays of color components, one to hold the red components of the colors, one for the green components, and one for the blue. When the user presses the mouse while using the smudge tool, the color components of the pixels in a 9-by-9 square around the mouse location are copied into the arrays. When the mouse moves, some of the color from the arrays is blended into the colors of the image pixels at the new mouse location, and, at the same time, some of the color from the image is blended into the arrays. That is, the arrays drop off some of the color that they are carrying and pick up some of the color from the new location. (If you think about it, you should see that something similar happens when you smear paint with your finger.)

To implement this idea, we need to be able to read the colors of pixels in the image, and we need to be able to write new colors to those pixels. It is fairly easy to write colors to individual pixels, using something called a *PixelWriter*. If `g` is a *GraphicsContext* for a canvas, you can get a *PixelWriter* for the canvas by calling

```
PixelWriter pixWriter = g.getPixelWriter();
```

Then, to set the color of the pixel at `(x,y)` in the canvas, you can simply call

```
pixWriter.setColor( x, y, color );
```

where `color` is of type *Color*. (Note that `x` and `y` are pixel coordinates; they are not subject to any transform that might have been set in the graphics context.)

If JavaFX had a similarly easy way to read pixel colors from a canvas, we would be all set. Unfortunately, it is not so simple. The reason, as I understand it, is technical: It turns out that drawing operations do not immediately draw to the canvas. Instead, for efficiency, a bunch of drawing operations are saved and sent to the graphics hardware in a batch. Ordinarily, a batch is sent only when the canvas needs to be redrawn on the screen. This means that if you simply read a pixel color from the canvas, the value that you get would not necessarily reflect all of the drawing commands that you have applied to the canvas. In order to read pixel colors, you have to do something that will force all of the drawing operations to complete. The only way I know to do that is by taking a "snapshot" of the canvas.

You can actually take a snapshot of any scene graph node. A snapshot returns a *WritableImage* that contains a picture of the node, after all pending operations have been applied. It's easy to take a picture of an entire node:

```
WritableImage nodePic = node.snapshot(null,null);
```

The *WritableImage* will contain a picture of the node as it would appear on the screen, and it can be used in the same way as a regular *Image* object. The parameters are of type *SnapshotParameters* and *WriteableImage*. If a non-null writable image is provided, it will be used for the image, as long as it is large enough to contain a picture of the node; this can be more efficient than creating a new writable image. The *SnapshotParameters* can be used to get more control over the picture that is produced. In particular, they can be used to specify that the snapshot should contain only a certain rectangle within the node.

To implement the smudge tool, we need to grab the pixels from a small, 9-by-9 rectangle within the canvas. To do that efficiently, we can provide *SnapshotParameters* that pick out just that rectangle. We can then read the colors of the pixels using a *PixelReader* for the *WritableImage*. There are a

lot of details, so I will just show you how it's done. In the program, I create a single *WritableImage*, a *PixelReader*, and a *SnapshotParameters* that will be used for all snapshots. Things are complicated a bit by the fact that some of the pixels in the snapshot might lie outside of the canvas, and I don't want to try to use color data for those non-existent pixels. Here is the code:

```
pixels = new WritableImage(9,9);        // a 9-by-9
writable image
pixelReader = pixels.getPixelReader();  // a PixelReader
for the writable image
snapshotParams = new SnapshotParameters();
```

When the uses presses the mouse, I need to take a snapshot of the 9-by-9 square in the canvas that surrounds the current mouse coordinates, `(startX,startY)`. And I need to copy the color data from the snapshot into the color component arrays, `smudgeRed`, `smudgeGreen`, and `smudgeBlue`:

```
snapshotParams.setViewport( new Rectangle2D(startX - 4,
startY - 4, 9, 9) );
    // (The SnapshotParameter's "viewport" is the
rectangle in the canvas
    //  that will be included in the snapshot.)
canvas.snapshot(snapshotParams, pixels);
int h = (int)canvas.getHeight();
int w = (int)canvas.getWidth();
for (int j = 0; j < 9; j++) {  // row in the snapshot
    int r = startY + j - 4;  // the corresponding row in
the canvas
    for (int i = 0; i < 9; i++) {  // column in the
snapshot
        int c = startX + i - 4;  // the corresponding
column in canvas
        if (r < 0 || r >= h || c < 0 || c >= w) {
                // The point (c,r) is outside the canvas.
                // A -1 in the smudgeRed array indicates
that the
                // corresponding pixel was outside the
canvas.
            smudgeRed[j][i] = -1;
        }
        else {
            Color color = pixelReader.getColor(i, j);
                // pixelReader gets color from the
snapshot
            smudgeRed[j][i] = color.getRed();
```

```
            smudgeGreen[j][i] = color.getGreen();
            smudgeBlue[j][i] = color.getBlue();
        }
    }
}
```

To blend the color from the color component arrays into a square of pixels around another point `(x,y)`, I need to take a new snapshot, this time of the 9-by-9 square of pixels surrounding `(x,y)`. Once we have that snapshot, we can do the blending calculations and write the new color back to the canvas, using the *PixelWriter* that writes to the canvas:

```
snapshotParams.setViewport( new Rectangle2D(x - 4, y - 4,
9, 9) );
canvas.snapshot(snapshotParams, pixels);
for (int j = 0; j < 9; j++) { // row in the snapshot
    int c = x - 4 + j;  // corresponding row in the canvas
    for (int i = 0; i < 9; i++) {  // column in the
snapshot
        int r = y - 4 + i;  // corresponding column in the
canvas
        if ( r >= 0 && r < h && c >= 0 && c < w &&
smudgeRed[i][j] != -1) {

               // PixelReader gets current pixel color
from the snapshot:
           Color oldColor = pixelReader.getColor(j,i);

               // Get a new color for the pixel by
blending current color
               // with the color components from the
arrays:
           double newRed = (oldColor.getRed()*0.8 +
smudgeRed[i][j]*0.2);
           double newGreen = (oldColor.getGreen()*0.8 +
smudgeGreen[i][j]*0.2);
           double newBlue = (oldColor.getBlue()*0.8 +
smudgeBlue[i][j]*0.2);

              // PixelWriter writes new pixel color to the
canvas:
           pixelWriter.setColor( c, r,
Color.color(newRed,newGreen,newBlue) );

              // Also blend some of the color from the
canvas into the arrays:
```

```
            smudgeRed[i][j] = oldColor.getRed()*0.2 +
smudgeRed[i][j]*0.8;
            smudgeGreen[i][j] = oldColor.getGreen()*0.2 +
smudgeGreen[i][j]*0.8;
            smudgeBlue[i][j] = oldColor.getBlue()*0.2 +
smudgeBlue[i][j]*0.8;
        }
    }
}
```

This is a fairly complex operation, but I hope it gives you an idea of how to work with pixels. Remember that you can write pixel colors to a canvas using a *PixelWriter*, but to read pixels from the canvas, you need to take a *Snapshot* of the canvas, and use a *PixelReader* to read pixel colors from the *WritableImage* that contains the snapshot.

13.2.6 Image I/O

The sample paint program ToolPaint.java has a "File" menu with commands for loading an image from a file into the canvas and for saving the image from the canvas into a file.

To load the image from a file, you first need to load it into an object of type *Image*. We have seen in Subsection 6.2.3 how to load an *Image* from a resource file and how to draw that image onto a canvas. Loading the image from a file is not much different:

```
Image imageFromFile = new Image( fileURL );
```

The parameter is a string that specifies the file location as a URL—essentially a path to the file, preceded by "file:". If *imageFile* is an object of type *File* representing a path to the file, you can say

```
Image imageFromFile = new Image( "file:" + imageFile );
```

Typically, you would let the user select the file using a *FileChooser* dialog (Subsection 11.2.3), and `imageFile` would simply be the selected *File* returned by that dialog.

Of course, the user might select a file that can't be read or that does not contain an image. The *Image* constructor does not throw an exception in that case. Instead, it sets an error condition in the image object that you can check using the boolean-valued function `imageFromFile.isError()`. If an error did occur, you can get the exception that caused the error by calling `imageFromFile.getException()`.

Once you have the image, and have checked that it was loaded without error, you can draw it to a canvas using a graphics context for the canvas. This command will scale the image to exactly fill the canvas:

```
g.drawImage( imageFromFile, 0, 0, canvas.getWidth(),
canvas.getHeight() );
```

This is all put together in a method `doOpenImage()` that is used in `ToolPaint` to load an image from a user-selected file:

```
private void doOpenImage() {
    FileChooser fileDialog = new FileChooser();
    fileDialog.setInitialFileName("");
    fileDialog.setInitialDirectory(
                            new File(
System.getProperty("user.home") ) );
    fileDialog.setTitle("Select Image File to Load");
    File selectedFile = fileDialog.showOpenDialog(window);
    if ( selectedFile == null )
        return;  // User did not select a file.
    Image image = new Image("file:" + selectedFile);
    if (image.isError()) {
        Alert errorAlert = new
Alert(Alert.AlertType.ERROR,
                "Sorry, an error occurred while\ntrying to
load the file:\n"
                      + image.getException().getMessage());
        errorAlert.showAndWait();
        return;
    }

canvasGraphics.drawImage(image,0,0,canvas.getWidth(),canva
s.getHeight());
}
```

To save the image from a canvas into a file, you must first get the image from the canvas by making a snapshot of the entire canvas. That can be done with

```
Image image = canvas.snapshot(null,null);
```

Unfortunately, at least in the current version, it seems that JavaFX does not itself include any support for saving images to files. For that, it depends on something from the older "AWT" GUI toolkit: the *BufferedImage* class from package `java.awt.image`. A *BufferedImage* represents an image stored in the computer's memory, similar to an *Image* in JavaFX. A JavaFX *Image* can easily be converted into a *BufferedImage*, using a static method in class *SwingFXUtils* from package `javafx.embed.swing`:

```
BufferedImage bufferedImage =
SwingFXUtils.fromFXImage(canvasImage,null);
```

The second parameter, which is `null` here, could be an existing *BufferedImage* to hold the image.

Once you have the *BufferedImage*, you can use a static method from class *ImageIO*, in package `javax.imageio`, to write it to a file:

```
ImageIO.write( bufferedImage, format, file );
```

The second parameter is a *String* that specifies the file format for the image. Images can be saved in several formats, including "PNG", "JPEG", and "GIF". `ToolPaint` always saves files in PNG format. The third parameter is of type *File*, and it specifies the file to be saved. `ImageIO.write()` will throw an exception if it is unable to save the file. (It can also fail without throwing an exception if it doesn't recognize the format.) Putting all this together gives the `doSaveImage()` method in `ToolPaint`:

```
private void doSaveImage() {
    FileChooser fileDialog = new FileChooser();
    fileDialog.setInitialFileName("imagefile.png");
    fileDialog.setInitialDirectory(
                     new File(
System.getProperty("user.home") ) );
    fileDialog.setTitle("Select File to Save. Name MUST
end with .png!");
    File selectedFile = fileDialog.showSaveDialog(window);
```

```
    if ( selectedFile == null )
        return;  // User did not select a file.
    try {
        Image canvasImage = canvas.snapshot(null,null);
        BufferedImage image =
SwingFXUtils.fromFXImage(canvasImage,null);
        String filename =
selectedFile.getName().toLowerCase();
        if ( ! filename.endsWith(".png")) {
            throw new Exception("The file name must end
with \".png\".");
        }
        boolean hasFormat =
ImageIO.write(image,"PNG",selectedFile);
        if ( ! hasFormat ) { // (this should never happen)
            throw new Exception( "PNG format not
available.");
        }
    }
    catch (Exception e) {
        Alert errorAlert = new
Alert(Alert.AlertType.ERROR,
               "Sorry, an error occurred while\ntrying to
save the image:\n"
                     + e.getMessage());
        errorAlert.showAndWait();
    }
}
```

Section 13.3

Complex Components and MVC

THERE IS A LOT MORE COMPLEXITY than we have seen so far lurking in the JavaFX API. However, a lot of that complexity works to your benefit as a programmer, since a lot of it is hidden in typical uses of JavaFX components. You don't have to know about the most complex details of controls in order to use them effectively in most programs.

JavaFX defines several component classes that are much more complex than those we have looked at, 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 and tables. To use these complex components effectively, it's helpful to know something about the Model-View-Controller pattern that is used as a basis for the design of many GUI components. That pattern is discussed later in this section.

There are a number of JavaFX controls that are not covered in this book. Some useful ones that you might want to look into include: *TabbedPane*, *SplitPane*, *Tree*, *ProgressBar*, and various specialized input controls such as *ColorPicker*, *DatePicker*, *PasswordField*, and *Spinner*.

We start this section with a short example of writing a custom control—something that you might consider when even the large variety of components that are already defined in JavaFX don't do quite what you want (or when they do too much, and you want something simpler).

13.3.1 A Simple Custom Component

JavaFX's standard component classes are usually all you need to construct a user interface. At some point, however, you might need something a bit different. In that case, you can consider writing your own component class,

by building on one of the components that JavaFX does provide or on the basic *Control* class that serves as the base class for all controls.

For example, suppose I have a need for a "stopwatch" control. 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 *Label*, but we want a *Label* that can respond to mouse clicks. We can get this behavior by defining a *StopWatchLabel* component as a subclass of the *Label* 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. In the mouse event handling method that responds to mouse clicks on the stopwatch, 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. We can use the method `System.currentTimeMillis()` to get the current time, in milliseconds, as a value of type long. When the timer is started, we can store the current time in an instance variable, `startTime`. When the timer is stopped, we can use the current time and the start time to compute the elapsed time that the timer has been running. The complete `StopWatch` class is very short:

```
import javafx.scene.control.Label;

/**
 * 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 Label {

    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 event
     * handler so the label can respond to clicks.
     */
    public StopWatchLabel() {
        super("  Click to start timer.  ");
        setOnMousePressed( e -> setRunning( !running ) );
    }


    /**
     * Tells whether the timer is currently running.
     */
    public boolean isRunning() {
        return running;
    }


    /**
     * Sets the timer to be running or stopped, and
changes the text that
     * is shown on the label.  (This method should be
called on the JavaFX
     * application thread.)
     * @param running says whether the timer should be
running; if this
     *    is equal to the current state, nothing is done.
     */
    public void setRunning( boolean running ) {
        if (this.running == running)
            return;
```

```
        this.running = running;
        if (running == true) {
                // Record the time and start the timer.
            startTime = System.currentTimeMillis();
            setText("Timing....");
        }
        else {
                // Stop the timer.  Compute the elapsed
time since the
                // timer was started and display it.
            long endTime = System.currentTimeMillis();
            double seconds = (endTime - startTime) /
1000.0;
            setText( String.format("Time: %1.3f seconds",
seconds) );
        }
    }

} // end StopWatchLabel
```

Don't forget that since *StopWatchLabel* is a subclass of *Label*, you can do anything with a *StopWatchLabel* that you can do with a *Label*. You can add it to a container. You can set its font, text color, max and preferred size, and CSS style. You can set the text that it displays (although this would interfere with its stopwatch function).

StopWatchLabel.java is not an application and cannot be run on its own. The very short program TestStopWatch.java shows a *StopWatchLabel*, and it sets several of the label's properties to improve the appearance.

13.3.2 The MVC Pattern

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 often needs 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 JavaFX 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, as well as listeners for other high-level events, such as those from a button or 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 JavaFX, even when the terms "model" and "view" are not used. The whole idea of observable properties (Subsection 13.1.1) is a way of implementing the idea of a model that is separate from the view, although when properties are used, the model

can be spread over many different objects. For the list and table controls that we will look at next, the model and view are somewhat more explicit.

13.3.3 ListView and ComboBox

A *ListView* is a control that represents a list of items that can be selected by the user. It is also possible to let the user edit items in the list. The sample program SillyStamper.java allows the user to select an icon (that is, a small image) in a *ListView* of icons. The user selects an icon in the list by clicking on it. The selected icon can be "stamped" onto a canvas by clicking on the canvas. Shift-clicking adds a larger version of the image to the canvas. (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 "star" icon selected in the list:

ListView is defined in package `javafx.scene.control`. It is a parameterized type, where the type parameter indicates what type of object can be displayed in the list. The most common type is probably `ListView<String>`, but in the sample program, it's `ListView<ImageView>`. A *ListView* can display *Strings* and *Nodes* directly. When used with objects of other types, the default is to display the string representation of the object, as returned by the `toString()` method (which is often not very useful).

The items in a `ListView<T>` are stored in an `ObservableList<T>`. The list of items is part of the model for the *ListView*. The list for a `listView` can be accessed as `listView.getItems()`. When items are added to this list or deleted from the list, the list view is automatically updated to reflect the change.

In the `SillyStamper` program, the list is static. That is, no changes are made to the list after it has been created, and the user cannot edit the list. The images for the icons are read from resource files, and each *Image* is wrapped in an *ImageView* (Subsection 6.4.1), which is added to the list of items in the *ListView*. The list is constructed in the following method, which is called in the program's `start()` method:

```
private ListView<ImageView> createIconList() {
    String[] iconNames = new String[] {
                // names of image resource file, in
directory stamper_icons
            "icon5.png", "icon7.png", "icon8.png",
"icon9.png", "icon10.png",
            "icon11.png", "icon24.png", "icon25.png",
"icon26.png",
            "icon31.png", "icon33.png", "icon34.png"
    };

    iconImages = new Image[iconNames.length];  // Used for
drawing the icons.

    ListView<ImageView> list = new ListView<>();

    list.setPrefWidth(80);
    list.setPrefHeight(100);

    for (int i = 0; i < iconNames.length; i++) {
        Image icon = new Image("stamper_icons/" +
iconNames[i]);
        iconImages[i] = icon;
        list.getItems().add( new ImageView(icon) );
    }

    list.getSelectionModel().select(0);  // Select the
first item in the list.

    return list;
}
```

This method sets a preferred width and height for the list. The default preferred size for a list seems to be about 200-by-400, no matter what it contains. The icon list needs a much smaller width. The preferred height is set to a small value, but in this program it will grow to fill the available space in the "right" position of the *BorderPane* that fills the window. The default preferred height would have forced the container to be bigger than I wanted.

The other point of interest is the use of the "selection model" of the list. The selection model is the part of the model that records which items are selected in the list. By default, at most one item can be selected in a list, which is the correct behavior for this program. However, it would be possible to allow multiple items to be selected at the same time by calling

```
list.getSelectionModel().setSelectionMode(SelectionMode.MU
LTIPLE);
```

With the default single selection mode, when the user selects an item, the previously selected item, if any, is deselected. The user can select an item by clicking it, and the program can set the selection by calling `list.getSelectionModel().select(index)`. The parameter is the index of the item in the list, with numbering starting from zero as usual. If `index` is -1, than no item in the list is selected. The selected item in the list is highlighted in the list view. The list view listens for changes in the selection model. When the user clicks an item, a mouse listener (which is a built-in part of the controller for the list) changes the selection model. When that happens, the list view is notified of the change and updates its appearance to reflect the fact that a different item is now selected. A program can find out which item is currently selected by calling

```
list.getSelectionModel().getSelectedIndex()
```

In SillyStamper.java, this method is used when the user clicks on the canvas, to find out which icon should be stamped into the picture.

The list in `SillyStamper.java` is not editable. Our second sample program, EditListDemo.java, shows two lists that can be edited by the user. One is a

list of strings and one is a list of numbers. The user can initiate editing of an item by double-clicking it, or simply by clicking it if it is already selected. The user finishes the edit by pressing return or by hitting the escape key to cancel the edit. Selecting another item in the list will also cancel the edit.

There are a few things that you need to do if you want the user to be able to edit the items in a *ListView*. The `list` must be made editable by calling

```
list.setEditable(true);
```

However, that by itself is not enough. The individual cells in the list must also be editable. The term "cell" refers to an area in a list that displays one item. A list cell is an object that is responsible for displaying the item and, optionally, implementing user editing of the item. Default list cells are not editable.

A *ListView* uses a **cell factory** to create list cell objects. The cell factory is another object that is responsible for creating cells. To get a different kind of cell, you have to provide a different cell factory for the list. (This follows something called the "factory pattern." By using a a factory object to create cells, the cells can be customized without changing the *ListView* source code. You simply have to install a new cell factory.)

Writing cell factories is not a trivial task, but JavaFX has several standard cell factories. If `listView` is of type `ListView<String>`, you can install a cell factory that makes editable cells simply by calling

```
listView.setCellFactory( TextFieldListCell.forListView()
);
```

The method `TextFieldListCell.forListView()` returns a cell factory that creates cells that can display and edit strings. When the cell is simply displaying the string, it actually uses a *Label* for the display. When the item is being edited, it is displayed as a *TextField*.

That's all you need to know to make an editable list of strings. There are other item types for which it is natural to display the item as a string and to use a *TextField* for editing the item. That includes numbers, single character values, dates, and times. However, when the items are not strings, there has to be some way to convert between items and their string representations. Again, JavaFX makes things reasonably easy in the common cases: It provides standard converters for all of the types mentioned above. For example, if `intList` is an editable `ListType<Integer>`, you can install an appropriate cell factory with

```
intView.setCellFactory(
           TextFieldListCell.forListView( new
IntegerStringConverter() ) );
```

The parameter to `forListView` is an object that can convert between integers and their string representations. As it happens I was not completely satisfied with the standard converter, since it didn't handle illegal inputs in the text field very well, so for the list of integers in the sample program EditListDemo.java, I wrote my own converter and used it create the cell factory for the list:

```
StringConverter<Integer> myConverter = new
StringConverter<Integer>() {
        // This custom string converter will convert a bad
input string to
        // null, instead of just failing.  And it will
display a null value
        // as "Bad Value" and an empty string value as 0.
```

```
    public Integer fromString(String s) {
            // Convert a string to an integer.
        if (s == null || s.trim().length() == 0)
            return 0;
        try {
            return Integer.parseInt(s);
        }
        catch (NumberFormatException e) {
            return null;
        }
    }

    public String toString(Integer n) {
            // Convert an integer to a String.
        if (n == null)
            return "Bad Value";
        return n.toString();
    }
};

listView.setCellFactory( TextFieldListCell.forListView(
myConverter ) );
```

As you can see, a *StringConverter* just needs to define two functions, `fromString()` and `toString()`.

Standard string converters can be found in package `javafx.util.converters`, and the factory class `TextFieldListCell` is in package `javafx.scene.control.cell`. There are also similar classes for use with cells in tables, which we will need below.

In addition to the lists themselves in the sample program, a few interesting things are done with labels and buttons, by using observable properties of the *ListView's* model, as discussed in Subsection 13.1.2. For example, there are labels that show the selected index and the selected item in the list. This is coded by binding the text property of the label with a property of the list's selection model:

```
Label selectedIndexLabel = new Label();
selectedIndexLabel.textProperty().bind(
                    listView.getSelectionModel()
                            .selectedIndexProperty()
```

```
                                .asString("Selected Index:
%d") );

Label selectedNumberLabel = new Label();
selectedNumberLabel.textProperty().bind(
                    listView.getSelectionModel()
                                .selectedItemProperty()
                                .asString("SelectedItem: %s")
);
```

And the button for deleting the selected item from the list should be enabled only when there is actually a selected item. This is coded by binding the disable property of the button:

```
deleteNumberButton.disableProperty().bind(
                    listView.getSelectionModel()
                                .selectedIndexProperty()
                                .isEqualTo(-1) );
```

In effect, the labels and button are being used as alternative views of the same selection model that is used by the list. This is a major feature of the MVC pattern: There can be multiple views of the same model object. The views listen for changes in the model. When the model changes, the views are notified of the change and update themselves to reflect the new state of the model.

There is also an "Add" button that adds an item to the list. This uses another part of the *ListView* model. The item is actually added to the *ObservableList* that holds the items. Since the *ListView* listens for changes in that list, it will be notified of the change and will update itself to show the new item in the visible on-screen view of the list. Beyond adding the item to the observable list, no other action is needed in the program to get the item to appear on the screen.

We can look briefly at another control, *ComboBox*, that bears a lot of similarities to *ListView*. In fact, a *ComboBox* is basically a list view in which only the selected item is normally visible. When the user clicks on a combo box, the full list of items pops up, and the user can select an item from the list. In fact, a *ComboBox* actually uses a *ListView* to show the items in the pop-up list. You have seen combo boxes used as pop-up menus in some

examples, starting with GUIDemo.java all the way back in Section 1.6. Like *ListView*, *ComboBox* is a parameterized type. Although other item types are supported (using cell factories and string converters), *String* is the most common type for the items. Creating a combo box and managing the selection is similar to working with *ListView*. For example,

```
ComboBox<String> flavors = new ComboBox<>();
flavors.getItems().addAll("Vanilla", "Chocolate",
"Strawberry", "Pistachio");
flavors.getSelectionModel().select(0);
```

It is possible to set a combo box to be editable (and no special cell factory is needed for that, as long as the items are strings). An editable combo box is like a strange combination of text field and list view. Instead of using a label to show the selected item, the combo box uses a text field. The user can edit the value in the text field, and the modified value becomes the selected value in the combo box. However, the original value of the modified item is not removed from the list; the new item is just added. And the new item does not become a permanent part of the list. In the above example, the effect of saying `flavors.setEditable(true)` is to let the user type in "Rum Raisin," or anything else, as the selected flavor, but doing so won't replace "Vanilla", "Chocolate", "Strawberry", or "Pistachio" in the list.

Unlike a *ListView*, a *ComboBox* emits an *ActionEvent* when the user selects a new item, either by selecting it from the pop-up menu or by typing a new value into an editable combo box and pressing return.

13.3.4 TableView

Like a *ListView*, a *TableView* control displays a collection of items to the user. However, tables are more complicated than lists. Table items are arranged in a grid of rows and columns. Each grid position is a cell of the table. Each column has a column header at the top, which contains a name for the column. A column contains a sequence of items, and working with one of the columns in a *TableView* is in many ways similar to working with a *ListView*.

`TableView<T>` is a parameterized type. The type *T* represents one row of the table; that is, an object of the type parameter, *T*, contains the data for one row. (It can contain additional data as well; the table can be a view of just some of the available data.) The data model for a `table` of type `TableView<T>` is an `ObservableList<T>`, and it can be accessed as `table.getItems()`. Each item represents one row, and you can add and delete complete rows of the table by adding and deleting elements of this list.

To define a table, it's not enough to provide the class that represents the rows of the table; you also have to say what data goes in each column of the table. A table column is described by an object of type `TableColumn<T,S>`, where the first type parameter, *T*, is the same as the type parameter of the table, and the second type parameter, *S*, is the data type for the items that appear in the cells of that column. The type `TableColumn<T,S>` indicates that the column displays items of type *S* derived from rows of type *T*. A table column object does not contain the items that appear in the column; they are contained in the objects of type *T* that represent the rows. But the table column object needs to specify how to **get** the item that appears in the column from the object that represents the row. That is specified by something called a "cell value factory." It is possible to write a cell value factory that computes any function of a row object, but the most common option is to use a *PropertyValueFactory* that simply picks out one of the properties of the row object.

Let's turn to an example. The sample program SimpleTableDemo.java shows an uneditable table that contains the fifty states of the United States with their capital cities and population:

State	Capital City	Population
New Mexico	Santa Fe	2059180
New York	Albany	19378104
North Carolina	Raleigh	9535475
North Dakota	Bismarck	672591
Ohio	Columbus	11536502
Oklahoma	Oklahoma City	3751354

The data for the rows of the table are in objects of type *StateData*, which is defined in the program as a public static nested class. (The class must be

public for use with a *PropertyValueFactory*; it does not need to be static or nested.) The data values for a row are properties of the class, in the sense that there is a getter method for each value. Properties defined by getter methods are sufficient for uneditable table values (though as we shall see, you need something different for editable table columns). Here is the class definition:

```
public static class StateData {
    private String state;
    private String capital;
    private int population;
    public String getState() {
        return state;
    }
    public String getCapital() {
        return capital;
    }
    public int getPopulation() {
        return population;
    }
    public StateData(String s, String c, int p) {
        state = s;
        capital = c;
        population = p;
    }
}
```

The table that displays the state data is created with

```
TableView<StateData> table = new TableView<>();
```

and an item for each of the fifty states is added to the table data model, which is stored in `table.getItems()`. Then the table column objects are created and configured and added to the table column model, which is stored in `table.getColumns()`:

```
TableColumn<StateData, String> stateCol = new
TableColumn<>("State");
stateCol.setCellValueFactory(
                new PropertyValueFactory<StateData,
String>("state") );
table.getColumns().add(stateCol);

TableColumn<StateData, String> capitalCol = new
```

```
TableColumn<>("Capital City");
capitalCol.setCellValueFactory(
                  new PropertyValueFactory<StateData,
String>("capital") );
table.getColumns().add(capitalCol);

TableColumn<StateData, Integer> populationCol = new
TableColumn<>("Population");
populationCol.setCellValueFactory(
                 new PropertyValueFactory<StateData,
Integer>("population") );
table.getColumns().add(populationCol);
```

The parameter to the *TableColumn* constructor is the text that appears in the header at the top of the column. As for the cell value factories, remember that a cell value factory needs to pull the value for a cell out of a row object of type *StateData*. For the first column, the type of data is *String*, so the property value factory takes an input of type *StateData* and outputs a property value of a type *String*. The output value is the property named "state" in the *StateData* object. More simply, the constructor

```
new PropertyValueFactory<StateData, String>("state")
```

creates a cell value factory that gets the value to be displayed in a cell by calling `obj.getState()`, where `obj` is the object that represents the row of the table which contains the cell. The other two columns are specified similarly.

That's about all you need to know to create a table in which the user cannot edit the contents of the cells. By default, the user will be able to change the column width by dragging a separator between two column headers. And the user can click a column header to sort the rows of the table into increasing or decreasing order according to the values in that column. Both capabilities can be turned off by setting properties of the *TableColumn* object, as we will do for the next example. The user can also rearrange the columns by dragging a table header to the left or right.

The sample program ScatterPlotTableDemo.java includes an example of a user-editable table. Each row in the table represents a point in the plane, and the two columns contain the x-coordinates and the y-coordinates of the

points. To add some interest, the program also shows the points on a canvas; the canvas is a "scatter plot" with a small crosshair at the location of each point. In the picture, one of the x-coordinates is being edited:

We need a data type to represent the rows of the table. It can be a simple class, with two properties `x` and `y` that hold the coordinates of a point. However, for an editable table column, we can't use simple properties defined by getters and setters. The problem is that an editable column expects the property for that column to be an *observable property* (see Section 13.1). More exactly, the class should follow the JavaFX pattern for observable properties: The values for the properties named `x` and `y` should be stored in observable property objects, and a point object, `pt`, should have instance methods `pt.xProperty()` and `pt.yProperty()`. These methods return the observable property objects so that they can be used to get and set the property value. Since the properties store double values, they can be of type *DoubleProperty*. The data class for the table is defined as:

```
public static class Point {
    private DoubleProperty x, y;
    public Point(double xVal, double yVal) {
        x = new SimpleDoubleProperty(this,"x",xVal);
        y = new SimpleDoubleProperty(this,"y",yVal);
    }
    public DoubleProperty xProperty() {
        return x;
```

```
    }
    public DoubleProperty yProperty() {
        return y;
    }
}
```

The class *DoubleProperty* itself is an abstract class. *SimpleDoubleProperty* is concrete subclass. whose constructor requires the object that contains the property, the name of the property, and the initial value of the property. Support for adding change and invalidation listeners to the property is already implemented in this class.

With the *Point* class in hand, we can create the table and add a few random points to it:

```
table = new TableView<Point>();
points = table.getItems();

for (int i = 0; i < 5; i++) { // add 5 random points to
the table
    points.add( new Point(5*Math.random(),
5*Math.random()) );
}
```

When a point is added to or deleted from the table, the canvas has to be redrawn. To implement that, we can add a listener to the list, `points`, that serves as the data model for the table:

```
points.addListener( (Observable e) -> redrawDisplay() );
```

(The type for the parameter, `e`, in the lambda expression is given as *Observable* because an observable list has two `addListener()` methods that require a one-parameter lamda expression. The explicit type for `e` makes it possible for the compiler to tell which one is meant here: We are adding an *InvalidationListener* rather than a *ListChangeListener*.)

We have arranged for the canvas to be redrawn whenever a point is added to or deleted from the table. However, it will not be redrawn when the user edits one of the points in the table. That is not a change in the list structure; it's a change inside one of the objects in the list. To respond to such changes, we could add listeners to both of the observable properties in each *Point*

object. In fact, that's what the table does so that it can respond to any changes in a point that might be made from outside the table. Since that seems a little extreme to me, the program actually listens for a different kind of event to handle cell edits. A table has an observable property named `editingCell` whose value is the cell that is currently being edited, if any, or is `null` if no cell is being edited. When the value of this property changes to `null`, it means that an editing operation has been completed. We can arrange for the canvas to be redrawn after every editing operation by adding a change listener to the `editingCell` property:

```
table.editingCellProperty().addListener( (o,oldVal,newVal)
-> {
    if (newVal == null) {
        redrawDisplay();
    }
});
```

To complete the definition of the table, we have to define the columns. As in any table, we need a cell value factory for each column, which can be created using a property value factory; this follows the same pattern as the previous example. But for an editable column, we also need a cell factory, just as we did for the editable list example above. Similarly to that example, the cell factory can be created using

```
TextFieldTableCell.forTableColumn(myConverter)
```

where `myConverter` is a `StringConverter<Double>`. Furthermore, for this program, it makes sense to stop the user from resizing the columns or sorting them. Here is the complete code for setting up one of the columns:

```
TableColumn<Point, Double> xColumn = new TableColumn<>("X
Coord");
xColumn.setCellValueFactory( new
PropertyValueFactory<Point, Double>("x") );
xColumn.setCellFactory(
TextFieldTableCell.forTableColumn(myConverter) );
xColumn.setSortable(false);
xColumn.setResizable(false);
xColumn.setPrefWidth(100);  // (Default size is too small)
table.getColumns().add(xColumn);
```

The only other thing that is needed to make the table work is to set it to be editable by calling `table.setEditable(true)`. As you can see, you need to do quite a lot to make a table work, especially if it's an editable table. However, tables are complex, and the code that JavaFX makes you write to set up a table is much less than you would need to implement a table directly.

By the way, you should pay attention to the way that we have exploited the MVC pattern in this program. The scatter plot is an alternative view of the same data model that is shown in the table. The data from the model is used when the canvas is redrawn, and that happens in response to events generated by changes in the model. It actually takes surprisingly *little* thought and work to make sure that the scatter plot is always a correct view of the data.

I urge you to study the source code, which is well-commented. In addition to seeing the full details of the *TableView*, you might be interested in looking at how transforms (Subsection 13.2.3) are used when drawing the scatter plot.

Section 13.4

Mostly Windows and Dialogs

ALL OF THE SAMPLE GUI PROGRAMS that we have looked at have used a single window. However, many real programs use multiple windows. In this section, we look at how to manage a multi-window application. We will also discuss dialog boxes—small popup windows that are mainly used for getting input from the user. As a bonus, you get to learn about *WebView*, a JavaFX control that implements much of the functionality of a web browser window.

13.4.1 Dialog Boxes

A **dialog**, or dialog box, is a window that is dependent on another window. That window is the "parent" or "owner" of the dialog box. If the parent window is closed, the dialog box is automatically closed as well.

A dialog box can be **modal** or **modeless**. When a modal dialog box is opened, its parent window is blocked. That is, the user cannot interact with the parent window until the dialog box is closed. There are also **application modal** dialog boxes, which block the entire application, not just one parent window. Many dialog boxes in JavaFX are application modal. Modal dialog boxes are often popped up during program execution to get input from the user or sometimes just to show a message to the user.

Modeless dialog boxes do not block interaction with their parent windows, but they will be closed automatically when the parent window closes. A modeless dialog box might be used to show an alternative view of data from the parent window, or to hold some controls that affect the window.

It is possible to make a *Stage* work as a dialog box, but many dialog boxes in JavaFX programs are created as objects belonging to the class *Dialog*, from

package `javafx.scene.control`, or to one of its subclasses. A *Dialog*, `dlg`, has two instance methods for showing the dialog: `dlg.show()` and `dlg.showAndWait()`. If the dialog is shown using `dlg.showAndWait()`, then it is modal. (A dialog opened using `showAndWait()` is not just modal; it is application modal.) The `showAndWait()` method does not return until the dialog box has been closed, so that any input from the dialog box will be available to the program immediately after the call to `showAndWait()`. If the dialog is shown using `dlg.show()`, on the other hand, the dialog is modeless. The `show()` method returns immediately, and the user can then use both regular windows and the dialog box, and can switch back and forth between them. Using a modeless dialog box is a little like parallel programming—you have to understand that two things are going on at the same time. We will consider only modal dialog boxes here.

`Dialog<T>` is a parameterized type. The type parameter represents the type of value that will be returned by the `showAndWait()` method. The return type is actually `Optional<T>`, representing a value of type *T* that might or might not be present. *Optional* is defined in package `java.util`. An *Optional* has a boolean method `isPresent()` to test whether the value is present, and a method `get()` that returns the value if one is present. An exception occurs if `get()` is called when no value is present. This just means that if you want to use the return value from `showAndWait()`, you should first use `isPresent()` to test whether a value was actually returned.

A dialog box will ordinarily contain one or more buttons for closing the dialog. Typical button names include "OK", "Cancel", "Yes", and "No". The most common buttons are represented by the enumerated type *ButtonType*, which has values including `ButtonType.OK`, `ButtonType.CANCEL`, `ButtonType.YES`, and `ButtonType.NO`. *ButtonType* is a common return type for a *Dialog*, representing the button that the user clicked to close the dialog. In that case, the dialog box is of type `Dialog<ButtonType>`.

The class *Alert* is a subclass of `Dialog<ButtonType>` that makes it easy to create dialogs that show a text message to the user, along with one or two buttons. This class was already used in Subsection 11.2.3, without much explanation, to show error messages to the user. An alert can be created with

```
Alert alert = new Alert( alertType, message );
```

The first parameter is of type *Alert.AlertType* which is a nested enumerated type with values including `Alert.AlertType.INFORMATION`, `Alert.AlertType.WARNING`, `Alert.AlertType.ERROR`, and `Alert.AlertType.CONFIRMATION`. Alerts of the first three types will have a single "OK" button and do nothing but show the message to the user; for these alerts, there is no reason to check the return value of `alert.showAndWait()`. A confirmation alert has an "OK" button and a "Cancel" button and is typically used to ask whether the user would like to continue with some potentially dangerous operation such as deleting a file; in this case, checking the return value is important. Here is a typical use:

```
Alert confirm = new Alert( Alert.AlertType.CONFIRMATION,
                           "Do you really want to delete " +
file.getName() );
Optional<ButtonType> response = confirm.showAndWait();
if ( response.isPresent() && response.get() ==
ButtonType.OK ) {
    file.delete();
}
```

In addition to buttons, a *Dialog* can have: a content area; header text that appears above the content area; a graphic that appears next to the header text, if there is any, or next to the content; and of course a title in the title bar of the dialog window. Usually the graphic, if any, would be a small icon image. For an *Alert*, the message goes in the content area. The other properties are set automatically, depending on the alert type, but they can be changed by calling methods from the *Dialog* class before showing the alert:

```
alert.setTitle( windowTitle );
alert.setGraphic( node );
alert.setHeaderText( headerText );
```

Any of the values can be null. The content can be set to an arbitrary scene graph node, replacing the text of the message, by calling

```
alert.getDialogPane().setContent( node );
```

but it would be more common to do that for a plain *Dialog* than for an *Alert*. Here are a couple of confirmation alerts, showing the various components of

the dialog window. For the dialog box on the right, the header text is null. Note, by the way, that to get multiline text in an alert, you have to include line feed ("\n") characters in the text.

For an example of a dialog that gets input from the user, the class *TextInputDialog* is a subclass of `Dialog<String>`, meaning that the return value of `showAndWait()` will be `Optional<String>`. A *TextInputDialog* contains a *TextField* where the user can enter a line of text, and it has an "OK" button and a "Cancel" button. The constructor has a parameter of type *String* that represents the initial content of the text input box. If you want to ask a question or show some message to the user, you can put it into the dialog header text. The return value, if present, will be the content of the input box. Note that the return value can be the empty string. If the user clicks "Cancel" or simply closes the dialog box, then the return value is not present. Here is a typical usage:

```
TextInputDialog getNameDialog = new TextInputBox("Fred");
getNameDialog.setHeaderText("Please enter your name.");
Optional<String> response = getNameDialog.showAndWait();
if (response.isPresent() && response.get().trim().length()
> 0) {
    name = response.get().trim();
}
else {
    Alert error = new Alert( Alert.AlertType.ERROR,
                                "Anonymous users are not
allowed!" );
    error.showAndWait();
    System.exit(1):
}
```

Since I find *Alert* and *TextInputDialog* a little cumbersome to use (especially since I tend to prefer dialogs without icons), I wrote my own utility class, SimpleDialogs.java, that defines several more convenient static methods for showing some common kinds of dialog. In particular,

- `SimpleDialogs.message(text)` — shows the text and an OK button. There is no return value. The text will be automatically wrapped, so that line feed characters are not needed in long messages. An optional second parameter can be included to specify the title for the dialog window.
- `SimpleDialogs.prompt(text)` — shows the text and a text input box, with an OK button and a Cancel button. This method returns a *String*, which will be the content of the input box if the user clicks OK, or will be null if the user cancels the dialog. The window title can be given as an optional second parameter, and the initial content of the text input box as an optional third parameter.
- `SimpleDialogs.confirm(text)` — shows the text along with "Yes", "No", and "Cancel" buttons. The return value is a *String* which will always be one of "yes", "no", or "cancel". Again, the window title can be given as an optional second parameter.

There are some other options, including a basic color chooser dialog, which you can learn about by reading the source code. The program TestDialogs.java lets the user try out the dialogs that are defined in *SimpleDialogs*.

13.4.2 WebView and WebEngine

In the rest of this section, we will look at a multi-window web browser program. Writing a web browser sounds complicated, and it is, but JavaFX makes it pretty easy, by doing most of the work in a couple of standard classes. The *WebView* class, in package `javafx.scene.control`, represents a control that can load and display a web page. It handles most web pages pretty well, including running JavaScript code. (JavaScript is the programming language that is used to program active web pages; it is not closely related to Java.) A *WebView* is basically the "view" part of the

Model-View-Controller pattern (Subsection 13.3.2). The actual work of loading and managing the web page is done by an object of type *WebEngine* that is part of the "controller." The "model" would be a data structure containing the content that is displayed in the web page. The model is created by the *WebEngine* when a page is loaded, and the *WebView* then displays the content.

A *WebView* must be placed into a window. The sample class BrowserWindow.java is a subclass of the standard window class, *Stage*, that represents a complete web browser window. A *BrowserWindow* contains a *WebView* along with a menu bar and some other controls. In particular, there is a text-input box where the user can type the URL for a web page and a "Load" button that the user can click to load the web page from that URL into the *WebView*. In addition, the *BrowserWindow* constructor can specify an initial URL to be loaded when the window is first opened.

A *WebView* has an associated *WebEngine* that can be obtained by calling `webEngine = webview.getEngine()`. Then, to load a web page, you can simply call

```
webEngine.load( urlString );
```

where `urlString` is a string containing the URL. (URLs were discussed in Subsection 11.4.1.) The `urlString` must start with a "protocol" such as "http:" or "https:"; in my program, I add "http://" to the front of the URL string, if the string does not already specify a protocol.

A new web page will also be loaded automatically if the user clicks a link on the page that is currently being displayed.

Web page loading is asynchronous. That is, `webEngine.load()` returns immediately, and the web page is loaded in a background thread. When the loading has completed, the web page is shown in the *WebView*. If the load fails for some reason, there is no automatic notification. However, you can get some information about what is going on by adding listeners to two observable *String* properties of the web engine: the *location* and the *title*. The `location` is the URL of the web page that is currently being displayed or loaded, while the `title` is the title of the current web page, which usually

appears in the title bar of the window that displays the web page. For example, the *BrowserWindow* class monitors the `title` property and sets its window title to match:

```
webEngine.titleProperty().addListener( (o,oldVal,newVal) -
> {
    if (newVal == null)
        setTitle("Untitled " +
owner.getNextUntitledCount());
    else
        setTitle(newVal);
});
```

(I will discuss the "owner" below.) It also monitors the `location` property and displays its value in a *Label* at the bottom of the window.

To monitor the progress of loads, you can also add a listener to the property `webEngine.getLoadWorker().stateProperty()`; see the BrowserWindow.java source code for an example.

I said above that a *WebView* (with its *WebEngine*) can run JavaScript code that occurs on web pages. That is not quite true. JavaScript has subroutines for popping up certain simple dialog boxes: an "alert" dialog to simply display a message to the user; a "prompt" dialog to ask the user a question and get back a response string; and a "confirm" dialog that shows a message with an "OK" and a "Cancel" button. For a confirm dialog, the return value is a boolean that tells whether the user dismissed the dialog by clicking "OK". By default, requests from JavaScript to show these dialogs are ignored by the *WebEngine*. However, it is possible to add event handlers to the web engine to respond to those requests. In *BrowserWindow*, I use dialog boxes from my *SimpleDialogs* class to respond to the events. For example, when JavaScript tries to pop up an alert dialog, the web engine generates an event of type AlertEvent. The data in the event object is the message that JavaScript wants to display. The *BrowserWindow* class responds by using `SimpleDialogs.message()` to display the message to the user:

```
webEngine.setOnAlert(
        evt -> SimpleDialogs.message(evt.getData(), "Alert
from web page") );
```

Handling prompt and confirm dialogs is a little different, since they must return a value. Here is what's done in the sample program:

```
webEngine.setPromptHandler( promptData ->
            SimpleDialogs.prompt( promptData.getMessage(),
                      "Query from web page",
promptData.getDefaultValue() ) );
webEngine.setConfirmHandler( str ->
            SimpleDialogs.confirm(str, "Confirmation
Needed").equals("yes") );
```

I haven't yet discussed the menu bar for a *BrowserWindow*. The menu bar contains a single menu, named "Window". That menu contains commands for opening new browser windows and for closing the current window. It also contains a list of browser windows that are currently open. The user can bring a different window to the front of the screen by selecting the window from that list. To understand how that works, you need to understand how *BrowserWindow* is used in a complete, multi-window program.

13.4.3 Managing Multiple Windows

The class *BrowserWindow* is not an *Application*. It cannot be run as a program; it represents just one window in a multi-window program. The executable class for the program is defined in WebBrowser.java. The class *WebBrowser*, like any JavaFX program, is a subclass of *Application*. It depends on BrowserWindow.java and SimpleDialogs.java, so you need all three Java files to run the program.

An *Application* has a `start()` method that is called by the system when the application begins. The method has a parameter of type *Stage* that represents the "primary window" for the program, but there is no requirement that the program actually use that window. The `start()` method in *WebBrowser* ignores the primary window and instead creates and shows a window of type *BrowserWindow*. That is the first window opened when the program is run. It is set to load the front page of the web version of this very textbook.

That could have been everything that `WebBrowser.java` needs to do — except for the "Window" menu, which contains a list of all open windows. That list is not part of the data for an individual window, so it has to kept somewhere else. In the web browser application, the application object, of type *WebBrowser*, maintains the list of open windows. There is only one application object in the program, so we have just one list of open windows. (Another possibility would have been to make the window list a `static` member variable in the *BrowserWindow* class, since `static` variables in a class are shared by all instances of the class.) The *WebBrowser* class has a `newBrowserWindow()` method for opening new windows. A *BrowserWindow* has an instance variable, `owner`, that refers to the *WebBrowser* application that opened the window. When the browser window wants to open a new window, it does so by calling `owner.newBrowserWindow(url)`, where the parameter, `url`, is the URL of the web site to be loaded by the new window, or is `null` to open an empty browser window.

By default in JavaFX, the size of a window is determined by the size of the *Scene* that it contains, and the window is centered on the screen. However, it is possible to set the size and location of a window before it is opened. For a multi-window program, it is not desirable for all of the windows to appear in exactly the same location. And it turns out that the default size for a *BrowserWindow* is probably too small for most computer screens. In the *WebBrowser* application, each window that is opened is offset a little from the location where the previous window was opened, and the size of the window depends on the size of the screen.

The class *Screen*, in package `javafx.stage` has a static method `Screen.getPrimary()` that returns an object containing information about the computer's main screen. And that object in turn has a method `Screen.getPrimary().getVisualBounds()` that returns a *Rectangle2D* representing the usable area of the main screen. This is used in the program's `start()` method to compute a size and location for the first window:

```
public void start(Stage stage) {  // (stage is not used)

    openWindows = new ArrayList<BrowserWindow>();  // List
of open windows.
```

```
    screenRect = Screen.getPrimary().getVisualBounds();

       // (locationX,locationY) will be the location of
the upper left
       // corner of the next window to be opened.  For the
first window,
       // the window is moved a little down and over from
the top-left
       // corner of the primary screen's visible bounds.
    locationX = screenRect.getMinX() + 30;
    locationY = screenRect.getMinY() + 20;

       // The window size depends on the height and width
of the screen's
       // visual bounds, allowing some extra space so that
it will be
       // possible to stack several windows, each
displaced from the
       // previous one.  (For aesthetic reasons, limit the
width to be
       // at most 1.6 times the height.)
    windowHeight = screenRect.getHeight() - 160;
    windowWidth = screenRect.getWidth() - 130;
    if (windowWidth > windowHeight*1.6)
        windowWidth = windowHeight*1.6;

       // Open the first window, showing the front page of
this textbook.

newBrowserWindow("https://math.hws.edu/javanotes/index.htm
l");

} // end start()
```

When a window is opened in the `newBrowserWindow()` method, its size and location are taken from the variables `windowWidth`, `windowHeight`, `locationX`, and `locationY`. And the values of `locationX` and `locationY` are modified so that the next window will be placed at a different location. In addition, the new window is added to the open window list. We also have to make sure that the window is removed from that list when it is closed. Fortunately, a window generates an event when it is closed. We can add an event handler to listen for that event, and the event handler can remove the window from the open window list. Here is the code for `newBrowserWindow()`:

```
void newBrowserWindow(String url) {
    BrowserWindow window = new BrowserWindow(this,url);
    openWindows.add(window); // Add new window to open
window list.
    window.setOnHidden( e -> {
            // Called when the window has closed.  Remove
the window
            // from the list of open windows.
        openWindows.remove( window );
        System.out.println("Number of open windows is " +
openWindows.size());
        if (openWindows.size() == 0) {
               // Program ends automatically when all
windows have been closed.
            System.out.println("Program ends because all
windows are closed");
        }
    });
    if (url == null) {
        window.setTitle("Untitled " +
getNextUntitledCount());
    }
    window.setX(locationX);         // set location and
size of the window
    window.setY(locationY);
    window.setWidth(windowWidth);
    window.setHeight(windowHeight);
    window.show();
    locationX += 30;    // set up location for NEXT window
    locationY += 20;
    if (locationX + windowWidth + 10 >
screenRect.getMaxX()) {
            // Window would extend past the right edge of
the screen,
            // so reset locationX to its original value.
        locationX = screenRect.getMinX() + 30;
    }
    if (locationY + windowHeight + 10 >
screenRect.getMaxY()) {
            // Window would extend past the bottom edge of
the screen,
            // so reset locationY to its original value.
        locationY = screenRect.getMinY() + 20;
    }
}
```

The *WebBrowser* class has a method `getOpenWindowList()` that returns the open window list. This method is used by a *BrowserWindow* when it constructs the "Window" menu. This is not done in a very efficient way: The menu is rebuilt each time it is shown. A menu emits an event when the user clicks the menu name, just before the menu is shown. The *BrowserWindow* registers a handler for that event with the Window menu. The event handler gets the open window list by calling `owner.getOpenWindowList()` and uses it to rebuild the menu before it appears on the screen. Here is the code, from the *BrowserWindow* class.

```
private void populateWindowMenu() {
    ArrayList<BrowserWindow> windows =
owner.getOpenWindowList();
    while (windowMenu.getItems().size() > 4) {
            // The menu contains 4 permanent items.  Remove
the other
            // items, which correspond to open windows and
are left
            // over from the previous time the menu was
shown.

windowMenu.getItems().remove(windowMenu.getItems().size()
- 1);
    }
    if (windows.size() > 1) {
            // Add a "Close All" command only if this is
not the only window.
        MenuItem item = new MenuItem("Close All and
Exit");
        item.setOnAction( e -> Platform.exit() );
        windowMenu.getItems().add(item);
        windowMenu.getItems().add( new SeparatorMenuItem()
);
    }
    for (BrowserWindow window : windows) {
        String title = window.getTitle(); // Menu item
text is the window title.
        if (title.length() > 60) {
                // Let's not use absurdly long menu item
texts.
            title = title.substring(0,57) + ". . .";
        }
        MenuItem item = new MenuItem(title);
        final BrowserWindow win = window; // (for use in a
lambda expression)
```

```
            // The event handler for this menu item will
bring the corresponding
            // window to the front by calling its
requestFocus() method.
        item.setOnAction( e -> win.requestFocus() );
        windowMenu.getItems().add(item);
        if (window == this) {
               // Since this window is already at the
front, the item
               // corresponding to this window is
disabled.
            item.setDisable(true);
        }
    }
}
```

And that just about covers things. As you can see, it's not very difficult to manage a multi-window application. And it is wonderfully easy to write a reasonably functional web browser in JavaFX. This has been a good example of building on existing classes. And we've also seen some nice new examples of working with events. With that, we have almost reached the end of this textbook. The final section will cover a few odds-and-ends of GUI programming.

Section 13.5

Finishing Touches

IN THIS FINAL SECTION, I will present a program that is more complex and a little 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, including some that apply to all programs, not just GUI programs.

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. Note that an even more capable Mandelbrot program can be found at

https://math.hws.edu/eck/js/mandelbrot/java/MB-java.html.

And there is a JavaScript version that will run in your web browser at

https://math.hws.edu/eck/js/mandelbrot/MB.html

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 for the Mandelbrot set 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. So, 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. That limit is `maxIterations`:

```
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 definitely 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 definitely **not** in the set. Here is a screenshot from the Mandelbrot Viewer program that uses this computation. The black region is the Mandelbrot set (except that not all black points are known to be definitely in the set):

When you use the program, you can "zoom in" on small regions of the plane. To do so, just click-and-drag the mouse on the image. This will draw a rectangular "zoom box" around part of the image, as shown in the illustration. 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. The image will first become "blocky" and then meaningless.)

You can 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, but it's just the arbitrary colors that are different. 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.)

The program has 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 `javafx.scene.control` 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 subpackages of this package, such as `edu.hws.eck.mdbfx`, 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 2.6.6 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 seven Java source code files. They can be found in the directory edu/hws/eck/mdbfx inside the `source` directory of the web site. (That is, they are in a directory named `mdbfx`, 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; this file is used by the program and will be discussed below. And there is an `examples` folder that contains resource files that are used with an "Examples" menu. Since the Mandelbrot Viewer uses JavaFX, you need to make sure that JavaFX is available to it when the program is compiled or run, as discussed in Section 2.6. For an Integrated Development Environment such as Eclipse, you should just have to add the `edu` directory to your project, provided that project has been configured to use JavaFX. If you want to work on the command line instead, you must be working in the directory that contains the `edu` directory, and you need to add JavaFX options to the `javac` and `java` commands. Let's say that you've defined `jfxc` and `jfx` commands that are equivalent to `javac` and `java` with JavaFX options included, as discussed in Subsection 2.6.7. Then, to compile the source code, use the command

```
jfxc  edu/hws/eck/mdbfx/*.java
```

or, if you use Windows,

```
jfxc  edu\hws\eck\mdbfx\*.java
```

The main application class for the program is defined by a class named *Main*. To run the program, use the command:

```
jfx  edu.hws.eck.mdbfx.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 the file MandelbrotCanvas.java. The *MandelbrotCanvas* class is a subclass of *Canvas* that can compute and display visualizations of the Mandelbrot set, as discussed above. The image that is shown is determined by the ranges of x and y values that are visible, the maximum number of iterations for the Mandelbrot algorithm, and the palette that is used to color the pixels. The values for these inputs come from elsewhere in the program; *MandelbrotCanvas* just calculates and displays the image, based on the inputs that it is given.

In addition to coloring the pixels of the image, the *MandelbrotCanvas* class uses a two-dimensional array to store the iteration count for each pixel in the image. As discussed above, the iteration count for a pixel is used, along with the palette, to determine the color to use for that pixel. If the palette is changed, the iteration counts are used to reset the color of each pixel without doing the whole Mandelbrot computation again. However, if the range of xy-values changes, or if the size of the window changes, all of the iteration counts must be recomputed. Since the computation can take 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 Subsection 12.2.3. The program uses one worker thread for each available processor. When the computation begins, the image is transparent and you see the gray background of the window. The full computation is broken into tasks, where each task computes one row of the image. After finishing the computation, the task will apply the appropriate

colors to the pixels in the assigned row. Since the canvas can only be modified in the JavaFX application thread, the task uses `Platform.runLater()` to make the changes. (See Subsection 12.2.1.) The user can continue to use the menus and even the mouse while the image is being computed.

The file MandelbrotPane.java represents the entire content of the Mandelbrot Viewer window. *MandelbrotPane* is a subclass of *BorderPane*. The center position of a *MandelbrotPane* holds a *MandelbrotCanvas*. In fact, a second, transparent, "overlay" canvas is stacked on top of the canvas that contains the image. When the user draws a "zoom box" with the mouse, the zoom box is actually drawn in the top canvas so that drawing it does not damage the image. (See Subsection 13.2.4.) The bottom position in the *MandelbrotPane* contains a *Label*, which serves as a "status bar" showing some information that might be interesting to the user. Finally, there is a menu bar at the top of the pane.

The menu bar for the program is defined in Menus.java, which defines a subclass of *MenuBar*. (See Subsection 6.6.2 for a discussion of menus and menu items.) All of the menu items in the *Menus* class, as well as the commands that they represent, are implemented by methods and nested subclasses in that class. Among the commands are file manipulation commands that use techniques from Subsection 11.2.3, Subsection 11.5.2, and Subsection 13.2.6. The "MaxIterations," "Palette," and "PaletteLength" menus each contain a group of *RadioMenuItems*. 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 event handler 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. There is also an "Examples" menu that contains settings for several sample views of pieces of the Mandelbrot set.

Much of the work of the program is done in *MandelbrotPane*. It installs MousePressed, MouseDragged, and MouseReleased event handlers on the overlay canvas to implement zooming the image. It also installs a

MouseMoved event handler that updates the status bar to display the xy-coordinates in the image that correspond to the mouse location. And a MouseExited event handler on the canvas is used to reset the status bar to read "Idle" when the mouse moves out of the canvas.

Furthermore, many menu commands are implemented by calling methods in the *MandebrotPane* class. For example, there is a `setLimits()` method that can be called to set the range of xy-values that are shown in the image, and there are methods for setting the palette, the palette length, and the value of `maxIterations`. Whenever one of these properties is changed, the Mandelbrot image has to be modified. When the palette or palette length is changed, *MandelbrotPane* computes a new palette of colors, and calls a method in *MandelbrotCanvas* to tell it to use the new palette. When the limits or maxIterations are changed, the image will have to be entirely recomputed. *MandelbrotPane* calls a `startJob()` method in *MandelbrotCanvas* to tell it to start a new background computation job. All of the work of setting up and managing the job is done in *MandelbrotCanvas*.

The *MandelbrotPane* that is being used in the program is a parameter to the *Menus* constructor, and the *Menus* object saves a copy in an instance variable, `owner`. Many of the menu commands operate on the *MandelbrotPane* or on the *MandelbrotCanvas* that it contains. In order to carry out these commands, the *Menus* uses its `owner` instance varialble to call methods from the pane object. As for the *MandelbrotCanvas*, *MandelbrotPane* has an instance method `getDisplay()` that returns a reference to the canvas that the pane contains. So, the *Menus* object can obtain a reference to the canvas by calling `owner.getDisplay()`. In previous examples in this book, 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.

MandelbrotPane, *MandelbrotCanvas*, and *Menus* are the major classes in the Mandelbrot Viewer program. Main.java defines the subclass of *Application* that must be executed to run the program. Its `start()` method just places a *MandelbrotPane* into the program's primary *stage*. (See Subsection 6.6.3.) It also has one other task, which is discussed below. The

program also contains three other classes. Two of them, from SetImageSizeDialog.java and SetLimitsDialog.java, define custom dialog boxes. I will not discuss them further; see Subsection 13.4.1. The final class is *I18n*, which 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 that are used in the program.

13.5.3 Events, Listeners, and Bindings

We have worked extensively with events and event listeners, and to a lesser extent with binding of observable properties. But it is interesting to look at how they are used in the Mandelbrot Viewer program, where there are several classes involved.

Let's start from the following fact: The *MandelbrotCanvas* class knows nothing about the *Menus* class, yet the menu bar contains menu items that seem to know what's happening in the canvas class. In particular, some of the menu items are disabled when a computation is in progress. Since the canvas does not call methods or use variables from the menu class, how can the menus know whether or not a computation is in progress in the canvas? The answer, of course, is events—or more exactly, a binding (Subsection 13.1.2). A *MandelbrotCanvas* has an observable boolean property named `working` that is true exactly when a computation is in progress. The menu items should be disabled when the value of that property is true. That can be set up in one line by binding the `disableProperty` of the menu item to the `workingProperty` of the canvas. For example, this is done in the `Menus` constructor for the `saveImage` menu item with the command

```
saveImage.disableProperty().bind(owner.getDisplay().workin
gProperty());
```

Here, `owner` refers to the *MandelbrotPane*, and `owner.getDisplay()` is the *MandelbrotCanvas* that it contains.

Similarly, there is a "Restore Previous Limits" menu command that will reset the range of the xy-values in the image to their values before the most recent change. The *Menus* class uses an instance variable, `previousLimits`, of type `double[]`, to remember the previous limits. But how does it get that information? When the user zooms in or out on the image, the change takes place entirely in the *MandelbrotPane* class. There has to be a way for the menus to be notified of the change. Again, the solution is an observable property, this time of type `ObjectProperty<double[]>`. The *Menus* constructor adds a *ChangeListener* to the `limitsProperty` from the *MandelbrotPane* class:

```
owner.limitsProperty().addListener( (o,oldVal,newVal) -> {
       // save old value of limitsProperty for use in
"Restore Previous Limits"
    previousLimits = oldVal;
    undoChangeOfLimits.setDisable( previousLimits == null
);
});
```

The point is that because events are used for communication, the *MandelbrotCanvas* and *MandelbrotPane* classes are not strongly coupled to the *Menus* class. In fact, they could be used without any modification in other programs that don't even use the same *Menus* class. The alternative to using events and binding would be to have the canvas and display classes call methods such as `limitsChanged()` or `computationStarted()` in the *Menus* class. This would be strong coupling: Any programmer who wanted to use *MandelbrotCanvas* would also have to use *Menus* as well, or would have to modify the canvas class so that it no longer refers to the *Menus*. Of course, not everything can be done with events and not all strong coupling is bad: The *MandelbrotPane* class refers directly to the *MandelbrotCanvas* class and cannot be used without it—but since the whole purpose of a *MandelbrotPane* is to hold a *MandelbrotCanvas*, the coupling is not a problem. (*MandelbrotCanvas*, on the other hand, could be used independently of *MandelbrotPane*.)

There is another interesting use of events in *MandelbrotPane*. The image canvas and overlay canvas are contained in a *StackPane* named `displayHolder`. When the user changes the size of the window, the size of

the `displayHolder` is changed to match. When that happens, the canvasses should be resized to fit the new display size, and a new computation should be started to compute a new image. The *MandelbrotPane* class installs listeners on the height and width properties of `displayHolder` to respond to changes in size. (See the discussion of canvas resizing in Subsection 13.1.2.) However when the user dynamically changes the size of the window, the size of `displayHolder` can change many times each second. It is a fairly big deal to set up and start a new image computation, not something I want to do many times in a second. If you try resizing the program's window, you'll notice that the canvas doesn't change size dynamically as the window size changes. The same image is shown 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 computed. Here is how this works.

The listeners that handle changes in the size of `displayHolder` are set up to call a method named `startDelayedJob()`:

```
displayHolder.widthProperty().addListener( e ->
startDelayedJob(300,true) );
displayHolder.heightProperty().addListener( e ->
startDelayedJob(300,true) );
```

The first parameter to `startDelayedJob()` is the number of milliseconds to wait before resizing the canvas and starting a new computation. (The second parameter merely says that the canvas does need to be resized before the computation is started.) To implement the delay, `startDelayedJob()` uses an object of type *Timer*, from package `java.util`. A *TimerTask* can be submitted to a *Timer* to be executed after a specified delay. Until the time when it is executed, it is possible to cancel a task. So, `startDelayedJob()` submits a task to the timer that will resize the canvas after 300 milliseconds. If `startDelayedJob()` is called again before that delay has expired, it cancels the previous task and submits a new one to the timer. The result is that no task actually gets executed until 300 milliseconds pass with no call to `startDelayedJob()` in that time.

(There is an option in the program to give a fixed size to the image. In that case, the size of the `displayHolder` should not change. It is possible that the image might not fill the window, leaving some of the gray background of the

window visible. It is also possible that the image might be too big for the window. In that case, scroll bars will appear that can be used to scroll the image. This is done by a *ScrollPane*, a container that contains one component and provides scroll bars for it if necessary. When the image size is fixed, the `displayHolder` is removed from the *MandelbrotPane* and placed into a *ScrollPane*, and the *ScrollPane* is then placed in the center position of the *MandelbrotPane*.)

13.5.4 A Few More GUI Details

To finish this textbook's look at GUI programming, I would like to mention a couple details that haven't quite fit into previous sections.

We have seen that resource files are files that are part of a program but do not contain program code. It is very easy to use image resource files with the *Image* class (Subsection 6.2.3). But any kind of data can be stored in resource files. For example, the Mandelbrot Viewer program has an "Examples" menu. When the user chooses a command from that menu, the parameters for some particular view of the Mandelbrot set are loaded into the program. Those parameters are stored in a resource file, which uses a simple XML text format. The question is, how does the program get access to the file? Unfortunately, it's not quite as easy as creating an *Image* from an image resource file.

Resources are 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 general resource is to obtain a *ClassLoader*. That class loader can then be used to locate the resource file. Every object has an instance method `getClass()`, which returns an object that represents the class of that object. The class object, in turn, has a method `getClassLoader()` that returns the *ClassLoader* that loaded the class. So, in an instance method of an object, you can say

```
ClassLoader classLoader = getClass().getClassLoader();
```

to get the class loader that you need. Another way to get a reference to a class object is to use `ClassName.class`, where *ClassName* is the name of any class. For example, in the Mandelbrot Viewer program, I could have used *Menus.class.getClassLoader()* to get the class loader for the program.

Once you have a class loader, you can use it to look up a resource file. The class loader will actually return a URL for a file, and you can read the data in the file from that URL. (URLs, and reading the data that they refer to, were discussed in Section 11.4.) To find the resource file, the class loader requires the path to the file, just as for an image resource file. The path includes the file name as well as any directories that you need to traverse to get to the file. The resource files that contain the Mandelbrot examples are inside a chain of directories: `edu/hws/eck/mdbfx/examples`, and the full path for one of the files whose name is "settings1.mdb" is `edu/hws/eck/mdbfx/examples/settings1.mdb`. Here's a command that gets a URL for that file:

```
URL resourceURL =

classLoader.getResource("edu/hws/eck/mdbfx/examples/settin
gs1.mdb");
```

Once you have the URL, you can open an *InputStream* to read the data from the file:

```
InputStream stream = resourceURL.openStream();
```

and that's exactly what's done to implement the "Examples" menu in the Mandelbrot Viewer program. By using an input stream, you can read any type of data from a resource file, and you can do anything you want with it. But

remember that for some kinds of data, such as images, Java has more convenient ways to load the data into a program.

My second, short topic is **accelerators** for menu items. An accelerator is a key or combination of keys on the keyboard that can be used to invoke the menu item, as an alternative to selecting the menu item with a mouse. A typical example is Control-S for saving a file. The class *KeyCombination*, from package `javafx.scene.input`, represents combinations of keys that can be used for accelerators. A *KeyCombination* can be created from a string such as "ctrl+S". The string contains items separated by plus signs. Each item except for the last represents a modifier key: "ctrl", "alt", "meta", "shift" and "shortcut". (The modfiers can use upper or lower case.) The "shortcut" modifier is special: On a Mac, it is equivalent to the command key, "meta"; on Windows and Linux, it is equivalent to the control key, "ctrl". So, by using "shortcut", you get the appropriate modifier for menu commands on the system where the program is running. The last item in a key combination string is one of the enumerated type constants from the *KeyCode* type. This will generally be an upper case letter, which represents one of the letter keys (it must be upper case), but it can also be a function key such as F9. (Not all keys will work.) For example, the string "ctrl+shift+N" represents holding down the control and shift keys and pressing the "N" key; the string "shortcut+S" represents holding down the appropriate modifier for the computer you are using and pressing the "S" key. The string that describes a key combination can be passed to the static method `KeyCombination.valueOf()` to make a key combination object that can be used to install an accelerator for a menu item. Here, for example, is the code from the Mandebrot Viewer program that adds an accelerator to the *MenuItem*, `saveImage`, for the "Save Image" command:

```
saveImage.setAccelerator(
KeyCombination.valueOf("shortcut+shift+S") );
```

Accelerators can be used with any kind of menu items, including *RadioMenuItem* and *CheckMenuItem*. In all cases when the user types the accelerator key combination, it has the same effect as selecting the item with the mouse. The accelerator for a menu item is ordinarily shown in the menu item, along with the text of the menu item. In the Mandelbrot Viewer

program, all of the commands in the "File" and "Control" menus have accelerators.

13.5.5 Internationalization

In the remainder of this section, we look at two topics the apply to all programs, not to GUI programming in particular. These are things that you are not likely to use in a small program written for your own use, but they are important for large applications.

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 Java 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. Periods are often used to divide words in the key string. 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. A properties file should use the UTF-8 charset (see Subsection 11.1.3).

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 value string, which 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 lookup 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, the translator can make sure that the file name is 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 the class 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 7.1.2.) 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.

13.5.6 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*. 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

application window closes, and we need a way to arrange for preferences to be saved whenever that happens. The solution is to use events: The `start()` method in *Main* registers an event handler with the window to listen for the event that is generated when the window closes. The handler for that event 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.mdbfx`, and its preferences are identified by the string "/edu/hws/eck/mdbfx". (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 prefnode = root.node(pathName);
```

where `pathname` is the string, such as "/edu/hws/eck/mdbfx", 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 contained the file most recently opened or saved by the user of the program. This is particularly satisfying, since the default behavior for a file dialog box is to start in the working 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.

Programming Exercises for Chapter 13

Exercise 13.1:

The folder nature-images contains several pictures of animals. (In the web site download, you can find that folder in the `chapter13` directory inside the `source` directory.) Write a "scratch off" program that could be used by a small child that works as follows: The program window starts by showing a large uniformly colored rectangle, with one of the animal pictures hidden behind it. As the user drags the mouse over the image, part of the colored overlay is scratched off, revealing the picture underneath. Here is what it should look like after part of the overlay has been removed:

You can implement this by using one canvas, containing the colored overlay, stacked on top of another canvas that contains the animal picture. (Stacked canvases were used in the sample program ToolPaint.java from Subsection 13.2.4.) To implement scratching off part of the overlay, just clear a small rect in the overlay canvas around the mouse location. The program should have some way to move on to the next picture. Another idea is to have

several different sizes of scratchers, so that an impatient child can use a giant one that will remove large swatches of color.

See the Solution

Exercise 13.2:

The *StopWatchLabel* component from Subsection 13.3.1 displays the text "Timing..." when the stopwatch is running. It would be nice if it displayed the elapsed time since the stopwatch was started. For that, you need to create an *AnimationTimer*. (See Subsection 6.3.5.) Add an *AnimationTimer* to the original source code, StopWatchLabel.java, to drive the display of the elapsed time in seconds.

See the Solution

Exercise 13.3:

Improve the program ToolPaint.java, from Section 13.2. You can any improvements you like, but here are some suggestions:

- Have separate menus for "Fill Color" and "Stroke Color".
- Make it possible for the user to draw shapes that are both filled and stroked. For example, add two new tools, "Stroked Filled Rect" and "StrokedFilledOval".
- Add a "Line Width" menu.
- Add keyboard accelerators for some commands (see Subsection 13.5.4).
- Make it possible to use a translucent fill color. A simple approach to this is to use a *CheckMenuItem* to select either fully opaque or 50% opaque fill. I don't advise trying to implement translucent stroke colors, since that's more difficult.
- Add an "Undo" command that will restore the image to what it was before the last time it was modified. This can be implemented by

making a copy of the image before you modify it. It's possible to have a multi-level undo, but that's harder and uses more memory.

Remember that the `ToolPaint` program requires SimpleDialogs.java.

See the Solution

Exercise 13.4:

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 XML version from the exercise. Use a *TableView* 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.

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 need to save the data when the user closes the window, which ends the program. To do that, you can add a listener to the program's *Stage* to handle the WindowHidden event. For an example of using that event, the Mandelbrot Viewer program from Section 13.5 uses it to save preferences when the program ends. For an example of creating an editable table, see ScatterPlotTableDemo.java.

(I suggest keeping things simple. You not being asked to write a real phone book application! The point is mostly to make an editable table. My program has text input boxes for name and number, and an "Add" button for adding a new entry containing the input in those boxes. My program always saves the data, whether or not the user has changed it. The interface will be poor: The

user has to double-click a cell to edit it and press return to finish the edit and save the new value. It is possible to make a table with a better editing interface, but to do that, you need to write a new *CellFactory* class for the table.)

See the Solution

Quiz on Chapter 13

Question 1:

What is an "observable property"?

Question 2:

Suppose that `input` is a *TextField* and that `label` is a *Label*. Suppose that you want the text on the label to always be the same as the text in the text field. Write **two** code segments to accomplish that, one using an event listener and one using property binding.

Question 3:

Describe the picture that is produced by the following code, where `canvas` is a *Canvas*:

```
GraphicsContext g = canvas.getGraphicsContext2D();
g.setFill(Color.WHITE);
g.fillRect(0,0,canvas.getWidth(),canvas.getHeight());
g.translate( canvas.getWidth()/2, canvas.getHeight()/2 );
g.rotate( 30 );
g.setFill(Color.RED);
g.fillRect(0,0,100,100);
```

Question 4:

Create a *LinearGradient* paint and use it to fill the rectangle drawn by `g.fillRect(100,100,300,200)`. The rectangle should look like this, light gray at the top and black at the bottom:

(The API for creating gradient paints is complicated. It's OK to look it up!)

Question 5:

Suppose that `g` is a *GraphicsContext*. Explain the purpose of the methods `g.save()` and `g.restore()`.

Question 6:

What does the acronym *MVC* stand for, and how does it apply to the *List* class?

Question 7:

What is the difference between a "modal" dialog box and a "modeless" dialog box?

Question 8:

The Java API includes some classes in a package named `org.w3c.dom`. Why such a funny package name?

Question 9:

Suppose that `closeItem` is a *MenuItem*. What is done by the following statement? (What is an "accelerator"?)

```
closeItem.setAccelerator( KeyCombination.valueOf("ctrl+W")
);
```

Question 10:

What is meant by *Internationalization* of a program?

See the Answers

Glossary

abstraction. Abstraction refers, in general, to the idea of providing a simplified or higher level interface to a complex system. It is closely related to the idea of a "black box." Abstraction makes it possible to understand or use a system, while ignoring some of the details of what actually goes on in the system. For example, control abstractions such as if statements and while loops are actually implemented in machine language by jump and conditional jump instructions, but it is possible—and easier—to use if statements and while loops without knowing anything about their machine language implementation. Control abstractions make it possible to implement algorithms on a higher level than machine language. Abstraction is a fundamental concept in computer science.

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.

ALU. Arithmetic Logic Unit. The ALU is the part of the CPU that performs arithmetic operations such as addition and subtraction and logical operations such as AND and OR.

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.

class invariant. A statement about the state of a class, or of an object created from that class, that is always true. Any method in a class, to be correct, must preserve the truth of all class invariants.

class variables and class methods. Alternative terms for "static variables" and "static methods", which are part of the class itself rather than of objects.

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.

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 *BorderPane*, that can contain other GUI 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 computer that actually performs calculations and carries out programs.

CSS. Cascading Style Sheets, a language that can be used to control the visual appearance of components in JavaFX or of elements on a web page.

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 method. A method in a Java interface that has an implementation. The default implementation is used in any class that implements the interface but does not override the method. Default methods are marked with the reserved word `default`. Not supported in Java 7 and earlier.

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 owner. Dialog boxes are usually popped up to get information from the user or to display a message to the user.

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.

factory method. A method, usually a static function, that returns an object. Factory methods are an alternative to constructors.

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.

fill. A drawing operation that applies a color (or other type of fill) to each of the pixels inside a shape.

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.

functional interface. A Java `interface` that defines only a single subroutine (where the term "interface" here means an interface that defines a Java type.)

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 JavaFX is an object belonging to the *GraphicsContext* 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.

immutable object. An immutable object cannot be modified after it is constructed, because all of its instance variables are `final`. (In my use of the term, an immutable object can contain pointers to other objects that are not immutable.)

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.

I/O stream. An abstraction representing a source of input data or a destination for output data. Java has four basic IO stream classes representing input and output of character and binary data. These classes form the foundation for Java's input/output API.

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 classes that implement generic data structures, including *ArrayList* and *TreeSet*, for example.

JavaFX. A toolkit for GUI applications, which was introduced as a more modern alternative to the Swing GUI toolkit. JavaFX is not a standard part of Java but is used in this textbook.

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.

lambda expression. A notation that defines an anonymous method. More precisely, a lambda expression is a kind of literal that represents a value whose type is given by a functional interface.

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 class. A class that is defined inside a method definition. Local classes are often anonymous, but that is not required.

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.

loop invariant. A statement such that, if the statement is true before a loop executes, then it will remain true after each execution of the loop, and therefore will still be true after the loop ends. Loop invariants can be a tool for proving correctness of loops.

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 (data structure). 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>*

map (stream operator). One of the fundamental operations on streams, defined as part of Java's stream API. A map operator applies a function to each element of a stream, producing a new stream consisting of the values output by the function.

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.

method reference. A notation for a lambda expression that represents a method that already exists in some class or object. A method reference uses the `::` operator, such as `Math::sqrt`.

module. In general, a component of a larger system that interacts with the rest of the system in a simple, well-defined, straightforward manner. In Java 9 and later, a module is a collection of Java packages, allowing explicit control of dependencies between different modules, and the standard Java packages have been divided into q set of modules.

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.

observable value. A value that generates an event when it is modified, so that observers of the value can be notified of the change and can react to it.

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, MacOS, 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 subpackages, such as `java.io` and `javafx.scene.control`.

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.

predicate. A function that outputs a boolean value. Predicates in Java can be represented by the parameterized functional interface `Predicate<T>`.

priority queue. A data structure representing a collection of items where each item has a "priority." A priority queue has operations *add* and *remove*. Items can be added in any order, but the *remove* operation always removes an item of minimal priority. (Some version of priority queue use maximum instead of minimum priority.)

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.

record. A simple data structure containing several data items, or fields, that are identified by name. The fields can be of different types. Java has record classes to represent (immutable) records.

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."

reduce (stream operator). One of the fundamental operations on a stream. A reduce operation combines all the elements from a stream in some

way, such as by summing them or finding their maximum, producing a final result.

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.

scene graph. In JavaFX, the hierarchical data structure that contains all the GUI components that are shown in a window.

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.

standard input. The standard source from which a program reads input data. It is represented by the object `System.in`. Usually, standard input comes from text typed by the user, but standard input can be "redirected" to read from another source, such as a file, instead.

standard output. The standard destination to which a program writes output text. It is represented by the object `System.out`. Usually, standard output is displayed to the user, but standard output can be "redirected" to write to another destination, such as a file, instead. There is also an object `System.err` that is meant for writing error messages.

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. In Java 8, an abstraction representing a stream of values that can be processed. A stream can be created from a *Collection*, an array, or some other data source. Java's stream API includes many predefined operations that can be applied to streams. The term "stream" also refers to I/O streams, which are used for input and output.

stroke. A drawing operation that applies a color (or other type of paint) to pixels along the boundary of a shape.

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.

Answers to End-Of-Chapter Quizzes

This section contains sample answer to the quizzes that occur at the end of each chapter.

Remember that, in many cases, many answers are possible. This is especially true for questions that ask you to write code!

Answers for Quiz on Chapter 1

Question 1:

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

Answer:

The CPU, or Central Processing Unit, is the active part of the computer. Its function is to execute programs that are coded in machine language and stored in the main memory (RAM) of the computer. It does this by repeating the fetch-and-execute cycle over and over; that is, it repeatedly fetches a machine language instruction from memory and executes it.

Question 2:

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

Answer:

An asynchronous event is one that occurs at an unpredictable time outside the control of the program that the CPU is running. It is not "synchronized" with the program. An example would be when the user presses a key on the keyboard or clicks the mouse button. (These events generate "interrupts" that cause the CPU to interrupt what it is doing and to take some action to handle the asynchronous event. After handling the event, the CPU returns to what it was doing before it was interrupted.)

Question 3:

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

Answer:

Compilers and interpreters have similar functions: They take a program written in some programming language and translate it into machine language. A compiler does the translation all at once. It produces a complete machine language program that can then be executed. An interpreter, on the other hand, just translates one instruction at a time, and then executes that instruction immediately. (Java uses a compiler to translate java programs into Java Bytecode, which is a machine language for the imaginary Java Virtual Machine. Java Bytecode programs are then executed by an interpreter.)

Question 4:

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

Answer:

Programs written in the machine language of a given type of computer can be directly executed by the CPU of that type of computer. High-level language programs must be translated into machine language before they can be executed. (Machine language instructions are encoded as binary numbers that are meant to be used by a machine, not read or written by people. High-level languages use a syntax that is closer to human language.)

Question 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?

Answer:

The Java compiler translates Java programs into a language called Java bytecode. Although bytecode is similar to machine language, it is not the machine language of any actual computer. A Java interpreter is used to run the compiled Java bytecode program. (Each type of computer needs its own Java bytecode interpreter, but all these interpreters interpret the same bytecode language.)

Question 6:

What is a *subroutine?*

Answer:

A subroutine is a set of instructions for performing some task that have been grouped together and given a name. Later, when that task needs to be performed, it is only necessary to call the subroutine by giving its name, rather than repeating the whole sequence of instructions.

Question 7:

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

Answer:

An object consists of some data together with a set of subroutines that manipulate that data. (An object is a kind of "module," or self-contained entity that communicates with the rest of the world through a well-defined interface. An object should represent some coherent concept or real-world object.)

Question 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.)

Answer:

A variable is a memory location that has been given a name so that it can easily be referred to in a program. The variable holds a value, which must be of some specified type. The value can be changed during the course of the execution of the program.

Question 9:

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

Answer:

A Java program can be compiled once into a Java Bytecode program. The compiled program can then be run on any computer that has an interpreter for the Java virtual machine. Other languages have to be re-compiled for each platform on which they are going to run. The point about Java is that it can be executed on many different types of computers without being recompiled.

Question 10:

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

Answer:

The Internet is a network connecting millions of computers around the world. Computers connected to the Internet can communicate with each other. The

Internet can be used for Email (which lets a user of one computer send a message to a user on another computer), file sharing (which is used to copy files between computers), and the World Wide Web (which lets a user view "pages" of information published on computers around the world).

Answers for Quiz on Chapter 2

Question 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.

Answer:

The syntax of a language is its grammar, and the semantics is its meaning. A program with a syntax error cannot be compiled. A program with a semantic error can be compiled and run, but gives an incorrect result. A missing semicolon in a program is an example of a syntax error, because the compiler will find the error and report it. If `N` is an integer variable, then the statement "`frac = 1/N;`" is probably an error of semantics. The value of `1/N` will be 0 for any `N` greater than 1. It's likely that the programmer meant to say `1.0/N`.

Question 2:

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

Answer:

A variable is a "box", or location, in the computer's memory that has a name. The box holds a value of some specified type. A variable declaration statement is a statement such as

```
int x;
```

which creates the variable x. When the computer executes a variable declaration, it creates the box in memory and associates a name (in this case,

x) with that box. Later in the program, that variable can be referred to by name.

Question 3:

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

Answer:

A "type" represents a set of possible values. When you specify that a variable has a certain type, you are saying what values it can hold. When you say that an expression is of a certain type, you are saying what values the expression can have. For example, to say that a variable is of type int says that integer values in a certain range can be stored in that variable.

Question 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?

Answer:

The only values of type boolean are `true` and `false`. Expressions of type boolean are used in places where true/false values are expected, such as the conditions in `while` loops and `if` statements.

Question 5:

Give the meaning of each of the following Java operators:

```
a)      ++

b)      &&
```

```
c)      !=
```

Answer:

The operator ++ is used to add 1 to the value of a variable. For example, "`count++`" has the same effect as "`count = count + 1`".

The operator && represents the word **and**. It can be used to combine two boolean values, as in "`(x > 0 && y > 0)`", which means, "x is greater than 0 **and** y is greater than 0."

The operation != means "is not equal to", as in "`if (x != 0)`", meaning "if x is not equal to zero.".

Question 6:

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

Answer:

An assignment statement computes a value and stores that value in a variable. Examples include:

```
x = 17;           // Assign a constant value to the
variable, x.
newRow = row;     // Copy the value from the variable, row,
                  //             into the variable, newRow.
ans = 17*x + 42;  // Compute the value of the expression
                  //     17*x + 42, and store that value in
ans.
```

An assignment statement is used to change the value of a variable as the program is running. Since the value assigned to the variable can be another variable or an expression, assignments statements can be used to copy data from one place to another in the computer, and to do complex computations.

Question 7:

What is meant by *precedence* of operators?

Answer:

If two or more operators are used in an expression, and if there are no parentheses to indicate the order in which the operators are to be evaluated, then the computer needs some way of deciding which operator to evaluate first. The order is determined by the precedence of the operators. For example, `*` has higher precedence than `+`, so the expression `3+5*7` is evaluated as if it were written `3+(5*7)`.

Question 8:

What is a *literal*?

Answer:

A literal is a sequence of characters used in a program to represent a constant value. For example, `'A'` is a literal that represents the value A, of type char, and `17L` is a literal that represents the number 17 as a value of type long. A literal is a way of writing a value, and should not be confused with the value itself.

Question 9:

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

Answer:

A class can be used to group together variables and subroutines that are contained in the class. These are called the static members of the class. For example, the subroutine `Math.sqrt` is a static member of the class called *Math*. Also, the `main` routine in any program is a static member of a class. The second possible purpose of a class is to describe and create objects. The class specifies what variables and subroutines are contained in those objects. In this role, classes are used in object-oriented programming (which we haven't studied yet in any detail.)

Question 10:

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

Answer:

Either statement will read a real number input by the user, and store that number in the variable, x. They would both read and return exactly the same value. The difference is that in the second statement (using `getlnDouble`), after reading the value the computer will continue reading characters from input up to and including the next carriage return. These extra characters are discarded.

Question 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` ?

Answer:

The reason is somewhat technical. The difference is due to the order of evaluation. When several `+` operators are used in a row, with no parentheses,

they are evaluated from left to right. `2 + 3 + "test"` is interpreted as `(2 + 3) + "test"`, so 2 and 3 are added together, giving 5, and then the 5 is concatenated onto the string `"test"`. On the other hand, `"test" + 2 + 3` is interpreted as `("test" + 2) + 3`, so the 2 is first concatenated onto the `"test"`, giving `"test2"`, and then the 3 is concatenated onto that. In the case of `"test" + 2 * 3`, the precedence rules for `+` and `*` come into play. Since `*` has higher precedence, this expression is interpreted as `"test" + (2 * 3)`, which evaluates to `"test6"`.

Question 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?

Answer:

(This was a real question from a real student.)

Although *String*, like int, double, and boolean, is a type name, there is a fundamental difference between *String* and the other types. *String* is the name of a **class**, while int, double, and boolean are primitive types. Eclipse colors all class names in the same way that it does *String*, and it uses a different color for the primitive types. (Although the difference between classes and primitive types might not seem very important to you now, it really is an important distinction and it's reasonable for Eclipse to use different colors for the two concepts.)

Question 13:

What is the purpose of an import directive, such as `import textio.TextIO` or `import java.util.Scanner`?

Answer:

The classes *TextIO* and *Scanner* are defined inside "packages" named *textio* and *java.util*. This means that you can't simply use the names "TextIO" or "Scanner" in a program and expect the computer to know what those names mean. To use those names in a program, they must be imported from the relevant packages.

Question 14:

Write a complete program that asks the user to enter the number of "widgets" they want to buy and the cost per widget. The program should then output the total cost for all the widgets. Use `System.out.printf` to print the cost, with two digits after the decimal point. You do not need to include any comments in the program.

Answer:

The number of widgets can be a variable of type int, and the cost per widget and total coast can be variables of type double. Using *TextIO*, the program can be written:

```
import textio.TextIO;
public class CostOfWidgets {
    public static void main(String[] args) {
        int numberOfWidgets;
        double costPerWidget;
        double totalCost;
        System.out.print("How many widgets do you want? ");
        numberOfWidgets = TextIO.getlnInt();
        System.out.print("How much does each widget cost? ");
        costPerWidget = TextIO.getlnDouble();
```

```
        totalCost = numberOfWidgets * costPerWidget;
        System.out.printf("The total cost is $%1.2f%n",
totalCost);
    }
}
```

Using *Scanner*, this would be:

```
import java.util.Scanner;
public class CostOfWidgets {
    public static void main(String[] args) {
        int numberOfWidgets;
        double costPerWidget;
        double totalCost;
        Scanner in = new Scanner(System.in);
        System.out.print("How many widgets do you want?
");
        numberOfWidgets = in.nextInt();
        System.out.print("How much does each widget cost?
");
        costPerWidget = in.nextDouble();
        totalCost = numberOfWidgets * costPerWidget;
        System.out.printf("The total cost is $%1.2f%n",
totalCost);
    }
}
```

Answers for Quiz on Chapter 3

Question 1:

What is an *algorithm*?

Answer:

An algorithm is an unambiguous, step-by-step procedure for performing a certain task, which is guaranteed to finish after a finite number of steps. (An algorithm is not the same thing as a program, but it can be the *idea* behind a program.)

Question 2:

Explain briefly what is meant by "pseudocode" and how is it useful in the development of algorithms.

Answer:

Pseudocode refers to informal descriptions of algorithms, written in a language that imitates the structure of a programming language, but without the strict syntax. Pseudocode can be used in the process of developing an algorithm with stepwise refinement. You can start with a brief pseudocode description of the algorithm and then add detail to the description through a series of refinements until you have something that can be translated easily into a program written in an actual programming language.

Question 3:

What is a *block statement?* How are block statements used in Java programs?

Answer:

A block statement is just a sequence of Java statements enclosed between braces, { and }. The body of a subroutine is a block statement. Block statements are often used in control structures. A block statement is generally used to group together several statements so that they can be used in a situation that only calls for a single statement. For example, the syntax of a while loop calls for a single statement: "while (condition) do statement". However, the statement can be a block statement, giving the structure: "while (condition) { statement; statement; ...}".

Question 4:

What is the main difference between a `while` loop and a `do..while` loop?

Answer:

Both types of loop repeat a block of statements until some condition becomes false. The main difference is that in a `while` loop, the condition is tested at the beginning of the loop, and in a `do..while` loop, the condition is tested at the end of the loop. It is possible that the body of a `while` loop might not be executed at all. However, the body of a `do..while` loop is executed at least once since the condition for ending the loop is not tested until the body of the loop has been executed.

Question 5:

What does it mean to *prime* a loop?

Answer:

The condition at the beginning of a while loop has to make sense even the first time it is tested, before the body of the loop is executed. To prime the loop is to set things up before the loop starts so that the test makes sense (that is, the variables that it contains have reasonable values). For example, if the test in the loop is "while the user's response is yes," then you will have to prime the loop by getting a response from the user (or making one up) before the loop.

Question 6:

Explain what is meant by an *animation* and how a computer displays an animation.

Answer:

An animation consists of a series of "frames." Each frame is a still image, but there are slight differences from one frame to the next. When the images are displayed rapidly one frame after another, the eye perceives motion. A computer displays an animation by showing one image on the screen, then replacing it with the next image, then the next, and so on.

Question 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.

Answer:

Here are two possible answers. Assume that `N` has been declared to be a variable of type int:

```
for ( N = 3;  N <= 36;  N = N + 3 ) {
    System.out.println( N );
}
```

```
or
        for ( N = 3;  N <= 36;  N++ ) {
            if ( N % 3 == 0 )
                System.out.println( N );
        }
```

Question 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!

}
```

Answer:

The problem already gives an outline of the program. The last step, telling the user whether the number is even or odd, requires an `if` statement to decide between the two possibilities.

```
import textio.TextIO;

public static void main (String[] args) {

   int n;  // the number read from the user

   TextIO.put("Type an integer: ");  // ask the use to
enter an integer

   n = TextIO.getInt();   // read the user's response

   if (n % 2 == 0)        // tell the user whether the
number is even or odd
      System.out.println("That's an even number.");
   else
```

```
        System.out.println("That's an odd number.");
}
```

Question 9:

Write a code segment that will print out two *different* random integers selected from the range 1 to 10. All possible outputs should have the same probability. Hint: You can easily select two random numbers, but you have to account for the fact that the two numbers that you pick might be the same.

Answer:

The code for selecting two random integers has to be wrapped in a loop that will end only when the two selected numbers are different. This can be done easily with a `do..while` loop. A `while` loop can also be used, but it must be "primed" in some way. Note that by using a loop to choose the numbers, we can be absolutely sure that after the loop ends, the two numbers are different. Here are three possible solutions:

```
(1)     int x,y; // Two random integers.
        do {
            x = (int)(10*Math.random() + 1);
            y = (int)(10*Math.random() + 1);
        } while (x == y);  // continue if x and y are the
same number.
        System.out.println(x + " " + y);

(2)     int x,y;  // Two random integers.
        x = y = 0;  // Give them junk value to "prime" the
loop to make sure it runs.
        while (x == y) {
            x = (int)(10*Math.random() + 1);
            y = (int)(10*Math.random() + 1);
        }
        System.out.println(x + " " + y);

(3)     int x,y; // Two random integers.
        x = (int)(10*Math.random() + 1);  // Pick x, and
keep its value.
        do { // The loop finds a y with a different value
from x.
```

```
            y = (int)(10*Math.random() + 1);
        } while (x == y);
        System.out.println(x + " " + y);
```

Question 10:

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.)

Answer:

The function `Integer.parseInt` can be used to convert the strings into integers. This function will throw an exception of type *NumberFormatException* if the conversion fails. A `try..catch` statement can catch this exception and print an error message. So, the code segment can be written:

```
try {
   int n1, n2;  // The values of s1 and s2 as integers.
   int sum;     // The sum of n1 and n2.
   n1 = Integer.parseInt(s1);
   n2 = Integer.parseInt(s2);
   sum = n1 + n2;   // (If an exception occurs, we don't
get to this point.)
   System.out.println("The sum is " + sum);
}
catch ( NumberFormatException e ) {
    System.out.println("Error!  Unable to convert strings
to integers.");
}
```

Question 11:

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);
    }
}
```

Answer:

The exact output printed by this program is:

```
2
4
8
16
32
64
```

(The hard part to get right is the 64 at the end. The value of N doubles each time through the loop. For the final execution of the loop, N starts out with the value 32, but N is doubled to 64 before it is printed.)

Question 12:

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);
   }
}
```

Answer:

The way to answer this question is to trace exactly what the program does, step-by-step. The output is shown below on the right. On the left is a table that shows the values of the variables x and y as the program is being executed.

```
 value of x    |   value of y                 OUTPUT
---------------|---------------             -------------
      5        |      1   [ before loop]
      4        |      4   [ = 1*4 ]               4
      3        |     12   [ = 4*3 ]               12
      2        |     24   [ = 12*2 ]              24
      1        |     24   [ = 24*1 ]              24
      0        |      0   [ = 24*0 ]              0
```

Question 13:

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;
}
```

Answer:

This is a tough one! The output from this program consists of the three lines:

```
R
M
N
```

As the `for` loop in this code segment is executed, `name.charAt(i)` represents each of the characters in the string "Richard M. Nixon" in succession. The statement `System.out.println(name.charAt(i))` outputs the single character `name.charAt(i)` on a line by itself. However, this output statement occurs inside an `if` statement, so only some of the characters are output. The character is output if `startWord` is true. This variable is initialized to true, so when `i` is 0, startWord is true, and the first character in the string, 'R', is output. Then, since 'R' does not equal ' ', `startWorld` becomes false, so no more characters are output until `startWord` becomes true again. This happens when `name.charAt(i)` is a space, that is, just *before* the 'M' is processed and again just before the 'N' is processed. In fact whatever the value of `name`, this `for` statement would print the first character in `name` and every character in `name` that follows a space. (It is *almost* true that this for statement prints the first character of each word in the string, but something goes wrong when the first character is a space or when there are two spaces in a row. What happens in these cases?)

Question 14:

Suppose that `numbers` is an array of type `int[]`. Write a code segment that will count and output the number of times that the number 42 occurs in the array.

Answer:

Use a `for` loop to go through the array and test each array element. If the value is 42, add 1 to a counter:

```
int count42;  // The number of 42s in the array
count42 = 0;
int i;  // loop control variable
for ( i = 0; i < numbers.length; i++ ) {
    if ( numbers[i] == 42 ) {
        count42++;
    }
}
```

```
System.out.println("There were " + count42 + " 42s in the
array.")
```

Question 15:

Define the **range** of an array of numbers to be the maximum value in the array minus the minimum value. Suppose that `raceTimes` is an array of type `double[]`. Write a code segment that will find and print the range of `raceTimes`.

Answer:

We need both the minimum and the maximum value in the array. We can compute both using one `for` loop.

```
double max;     // maximum value from the array
double min;     // minimum value from the array
double range;   // the range of the array, max - min
int i;          // for-loop variable
max = min = raceTimes[0];  // start with both equal to the
first element
for ( i = 1; i < raceTimes.length; i++ ) {
    if ( raceTimes[i] > max )
        max = raceTimes[i];
    if ( raceTimes[i] < min )
        min = raceTimes[i];
}
range = max - min;
System.out.println("The range is " + range);
```

Answers for Quiz on Chapter 4

Question 1:

A "black box" has an interface and an implementation. Explain what is meant by the terms *interface* and *implementation*.

Answer:

The interface of a black box is its connection with the rest of the world, such as the name and parameters of a subroutine or the dial for setting the temperature on a thermostat. The implementation refers to internal workings of the black box. To use the black box, you need to understand its interface, but you don't need to know anything about the implementation.

Question 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?

Answer:

The contract of a subroutine says what must be done to call the subroutine correctly and what it will do when it is called. It is, in short, everything a programmer needs to know about the subroutine in order to use it correctly. (It does not include the "insides," or implementation, of the subroutine.)

The syntactic component of a subroutine's contract includes the name of the subroutine, the number of parameters, and the type of each parameter. This is the information needed to write a subroutine call statement that can be successfully compiled. The semantic component of the contract specifies the

meaning of the subroutine, that is, the task that the subroutine performs. It might also specify limitations on what parameter values the subroutine can process correctly. The semantic component is not part of the program. It is generally expressed in comments.

Question 3:

Briefly explain how subroutines can be useful in the top-down design of programs.

Answer:

Top-down refers to starting from the overall problem to be solved, and breaking it up into smaller problems that can be solved separately. When designing a program to solve the problem, you can simply make up a subroutine to solve each of the smaller problems. Then you can separately design and test each subroutine.

Question 4:

Discuss the concept of *parameters*. What are parameters for? What is the difference between *formal parameters* and *actual parameters*?

Answer:

Parameters are used for communication between a subroutine and the part of the program that calls the subroutine. If a subroutine is thought of as a black box, then parameters are part of the interface to that black box. Formal parameters are found in the subroutine definition. Actual parameters are found in subroutine call statements. When the subroutine is called, the values of the actual parameters are assigned to the formal parameters before the body of the subroutine is executed.

Question 5:

Give two different reasons for using named constants (declared with the `final` modifier).

Answer:

A constant has a meaningful name, which makes the program easier to read. It's easier to understand what a name like `INTEREST_RATE` is for than it is to figure out how a literal number like 0.07 is being used.

A second reason for using named constants is that it's easy to modify the value of the constant if that becomes necessary. If a literal value is used throughout the program, the programmer has to track down each occurrence of the value and change it. When a constant is used correctly, it is only necessary to change the value assigned to the constant at one point in the program.

A third reason is that using the `final` modifier protects the value of a variable from being changed. This is especially important for member variables that are accessible from outside the class where they are declared.

Question 6:

What is an API? Give an example.

Answer:

An API is an Applications Programming Interface. It is the interface to a "toolbox" of subroutines that someone has written. It tells you what routines are available, how to call them, and what they do, but it does not tell you how the subroutines are implemented. An example is the standard Java API which describes the interfaces of all the subroutines in all the classes that are available in such packages as `java.lang` and `java.util`.

Question 7:

What might the following expression mean in a program?

```
(a,b) -> a*a + b*b + 1
```

Answer:

The operator "`->`" means that this is a "lambda expression," that is, an anonymous function. There must be a functional interface that specifies the type of this lambda expression. It might, for example, be a functional interface that defines a function `double f(double x, double y)`. In any case, the function must take two numeric parameters (since the expression `a*a+b*b+1` would only be defined when *a* and *b* are numeric), and it must return a numeric value. (Note that this expression can only be used in a context where the compiler can deduce which functional interface applies.)

Question 8:

Suppose that *SupplyInt* is a functional interface that defines the method `public int get()`. Write a lambda expression of type *SupplyInt* that gets a random integer in the range 1 to 6 inclusive. Write another lambda expression of type *SupplyInt* that gets an int by asking the user to enter an integer and then returning the user's response.

Answer:

A lambda expression of type *SupplyInt* has no parameters and must compute and return a value of type int. For the first expression, the int value can be given as `(int)(1+Math.random()*6)`, so the lambda expression takes the form

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

For the second lambda expression, at least two statements are needed, one to ask the user for an integer and one to read the response. The two statements must be enclosed between braces, and a `return` is needed to return the integer:

```
() -> {
    System.out.print( "Please enter an integer: " );
    return TextIO.getlnInt();  // Read an integer and
return it.
}
```

(This assumes that *TextIO* is available to the program.)

Question 9:

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

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

Answer:

The subroutine could be written as follows:

```
static void stars(int numberOfStars) {
     // output a line containing the specified number of
stars
   for (int i = 0; i < numberOfStars; i++) {
       System.out.print('*');
   }
   System.out.println();  // output carriage return after
the *'s
}
```

Question 10:

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.

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

Answer:

The `main()` routine can use a `for` loop that calls the `stars()` subroutine ten times, once to produce each line of output. (An occasional beginner's mistake in this problem is to rewrite the body of the subroutine inside the `main()` routine, instead of just calling it by name.) Here is the main routine—which would, of course, have to be put together with the subroutine in a class in order to be used.

```
public static void main(String[] args) {
    int line;  // Line number, and also the number of
stars on that line.
    for ( line = 1;  line <= 10;  line++ ) {
        stars( line );
    }
}
```

Question 11:

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.

Answer:

The returned value will be of type int. The function simply uses a `for` loop to look at each character in the string. When the character in the string matches the parameter value, it is counted.

```
static int countChars( String str, char searchChar ) {
      // Count the number of times searchChar occurs in
      // str and return the result.
    int i;     // A position in the string, str.
    char ch;   // A character in the string.
    int count; // Number of times searchChar has been
found in str.
    count = 0;
    for ( i = 0;  i < str.length();  i++ ) {
        ch = str.charAt(i);  // Get the i-th character in
str.
        if ( ch == searchChar )
           count++;
    }
    return count;
}
```

Question 12:

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.

Answer:

I'll call the subroutine `smallest` and the three parameters `x`, `y`, and `z`. The value returned by the subroutine has to be either `x` or `y` or `z`. The answer will be `x` if `x` is less than or equal to both `y` and `z`. The correct syntax for checking this is "`if (x <= y && x <= z)`". Similarly for `y`. The only other remaining possibility is `z`, so there is no necessity for making any further test before returning `z`. (In fact, doing so would be an error in Java, since with no "else" clause in the `if` statement, the compiler cannot determine that the function definitely returns a value in all possible cases.)

```
static int smallest(int x, int y, int z) {
   if (x <= y && x <= z) {
      return x;
   }
   else if (y <= x && y <= z) {
      return y;
   }
   else
      return z;
}
```

Note: Since a return statement causes the computer to terminate the execution of a subroutine anyway, this could also be written as follows, without the `elses`:

```
static int smallest(int x, int y, int z) {
   if (x <= y && x <= z) {
      return x;
   }
   if (y <= x && y <= z) {
      return y;
   }
   return z;
}
```

Question 13:

Write a function that finds the average of the first `N` elements of an array of type double. The array and `N` are parameters to the subroutine.

Answer:

Note that the array must be passed as a parameter of type `double[]`, and that the value returned by the function will be a double. For the value of `N` to make sense, it should be in the range 1 up to the length of the array. My answer throws an *IllegalArgumentException* if `N` is not in this range:

```
static double average( double[] numbers, int N ) {
    if ( N < 1 ) {
        throw new IllegalArgumentException("Can't find an
average of " +
```

```
                        N + " numbers.");
    }
    if ( N > numbers.length ) {
        throw new IllegalArgumentsExcpetion( N +
                        " is more than the length of the
array." );
    }
    double sum = 0;  // the sum of the N numbers
    for ( int i = 0; i < N; i++ ) {
        sum = sum + numbers[i];  // add the i-th number to
the sum
    }
    return sum/N;  // Return the average as the value of
the function.
}
```

Question 14:

Explain the purpose of the following function, and explain how it works:

```
static int[] stripZeros( int[] list ) {
    int count = 0;
    for (int i = 0; i < list.length; i++) {
        if ( list[i] != 0 )
            count++;
    }
    int[] newList;
    newList = new int[count];
    int j = 0;
    for (int i = 0; i < list.length; i++) {
        if ( list[i] != 0 ) {
            newList[j] = list[i];
            j++;
        }
    }
    return newList;
}
```

Answer:

This function makes a copy of its parameter, except that it leaves out all the elements of `list` that are equal to zero. It builds a new array that contains all the non-zero elements of `list`, and it returns that array as the value of the

function. (Note that this is an example of using an array type as the return type of a function.)

The function creates a new array to be the return value. But to do that, it must know how long to make the array. The first five lines of the function definition count the number of non-zero elements in `list`. This is how many spaces we need in the new array, so `count` is used as the length when the new array is created. The remainder of the function goes through the original `list` and copies elements into `newList`. An element is copied only if it is non-zero. We have to keep track of how many spaces in `newList` have been filled so far. That's what `j` is for. This is the "partially filled array" pattern from Subsection 3.8.4.

Answers for Quiz on Chapter 5

Question 1:

Object-oriented programming uses *classes* and *objects*. What are classes and what are objects? What is the relationship between classes and objects?

Answer:

When used in object-oriented programming, a class is a factory for creating objects. (We are talking here about the non-static part of the class.) An object is a collection of data and behaviors that represent some entity (real or abstract). A class defines the structure and behaviors of all entities of a given type. An object is one particular "instance" of that type of entity. For example, if *Dog* is a class, then a particular dog named *Lassie* would be an object of type *Dog*.

Question 2:

Explain carefully what *null* means in Java, and why this special value is necessary.

Answer:

When a variable is of object type (that is, declared with a class or interface as its type rather than one of Java's primitive types), the value stored in the variable is not an object. Objects exist in a part of memory called the heap, and the variable holds a *pointer* or *reference* to the object. Null is a special value that can be stored in a variable to indicate that it does not actually point to any object.

Question 3:

What is a *constructor?* What is the purpose of a constructor in a class?

Answer:

A constructor is a special kind of subroutine in a class. It has the same name as the name of the class, and it has no return type, not even `void`. A constructor is called with the `new` operator in order to create a new object. Its main purpose is to initialize the newly created object, but in fact, it can do anything that the programmer wants it to do.

Question 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.)

Answer:

This statement creates a new object belonging to the class `Kumquat`, and it stores a reference to that object in the variable `fruit`. More specifically, when the computer executes this statement, it allocates memory to hold a new object of type `Kumquat`. It calls a constructor, which can initialize the instance variables of the object as well as perform other tasks. A reference to the new object is returned as the value of the expression "`new Kumquat()`". Finally, the assignment statement stores the reference in the variable, `fruit`. So, `fruit` can now be used to access the new object.

Question 5:

What is meant by the terms *instance variable* and *instance method*?

Answer:

Instance variables and instance methods are non-static variables and methods in a class; that is, their definitions in the class are **not** marked with the "static" modifier. This means that they do not belong to the class itself. Instead, they specify what variables and methods are in an object that belongs to that class. That is, the class contains the source code that defines instance variables and instance methods, but actual instance variables and instance methods are contained in objects, at least logically. (Such objects are called "instances" of the class.) Thus, instance variables and instance methods are the data and the behaviors of objects.

Question 6:

Explain what is meant by the terms *subclass* and *superclass*.

Answer:

In object oriented programming, one class can inherit all the properties and behaviors from another class. It can then add to and modify what it inherits. The class that inherits is called a subclass, and the class that it inherits from is said to be its superclass. In Java, the fact that `ClassA` is a subclass of `ClassB` is indicated in the definition of `ClassA` as follows:

```
class ClassA extends ClassB {...}
```

Question 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;
}
```

Answer:

To make a variable private, just add the word "private" in front of each declaration. We need two methods for each variable. One of them returns the value of the variable. The other provides a new value for the variable. The names for these methods should follow the usual naming convention for getter and setter methods. (Note that my setter methods use the special variable `this` so that I can use the same name for the parameter of the method as is used for the instance variable. This is a very common pattern.)

```
public class Player {
   private String name;
   private int score;
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;  // ("this.name" refers to the
instance variable)
   }
   public int getScore() {
      return score;
   }
   public void setScore(int score) {
      this.score = score;
   }
}
```

Question 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.

Answer:

If a class is not declared to `extend` any class, then it automatically extends the class *Object*, which is one of the built-in classes of Java. So in this case, *Player* is a direct subclass of *Object*. The *Object* class defines a

`toString()` method, and the *Player* class inherits this `toString()` method from *Object*. The methods and member variables in a class include not just those defined in the class but also those inherited from its superclass. (However, the inherited `toString()` method will not produce a useful string representation of a *Player*; to get that, you would have to override `toString()` in the *Player* class.)

Question 9:

Explain the term *polymorphism.*

Answer:

Polymorphism refers to the fact that different objects can respond to the same method in different ways, depending on the actual type of the object. This can occur because a method can be overridden in a subclass. In that case, objects belonging to the subclass will respond to the method differently from objects belonging to the superclass.

(Note: If B is a subclass of A, then a variable of type A can refer to either an object of type A or an object of type B. Let's say that `var` is such a variable and that `action()` is a method in class A that is redefined in class B. Consider the statement "var.action()". Does this execute the method from class A or the method from class B? The answer is that there is no way to tell! The answer depends on what type of object `var` refers to, a class A object or a class B object. The method executed by `var.action()` depends on the actual type of the object that `var` refers to **at run time**, not on the type of the variable `var`. This is the real meaning of polymorphism.)

Question 10:

Java uses "garbage collection" for memory management. Explain what is meant here by garbage collection. What is the alternative to garbage collection?

Answer:

The purpose of garbage collection is to identify objects that can no longer be used, and to dispose of such objects and reclaim the memory space that they occupy. If garbage collection is not used, then the programmer must be responsible for keeping track of which objects are still in use and disposing of objects when they are no longer needed. If the programmer makes a mistake, then there is a "memory leak," which might gradually fill up memory with useless objects until the program crashes for lack of memory.

Question 11:

What is an *abstract class*, and how can you recognize an abstract class in Java?

Answer:

An abstract class is one that cannot be used to create objects. It exists only as a basis for making subclasses, and it expresses all the properties and behaviors that those subclasses have in common. In Java, a class can be marked with the modifier `abstract` to make it abstract. For example,

```
abstract public class Vehicle { ...
```

It will then be a syntax error to try to call a "`new Vehicle`" constructor. (Note: Only a class that has been marked as `abstract` can contain abstract instance methods.)

Question 12:

What is `this`?

Answer:

"`this`" is a special variable in Java, which does not have to be declared. Java makes it available automatically in instance methods and constructors. It holds a reference to the object that is being constructed or that contains the instance method that is being executed (or, in terms of messages, the object that received the message that is being processed). It provides a way to refer to "this object." If `x` is an instance variable, it can also be referred to as `this.x` within the same class. If `doSomething()` is an instance method, it can also be called as `this.doSomething()` within the same class. (Personally, I would be happier with Java if it required the use of "this" instead of using it implicitly.)

Question 13:

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`.

Answer:

Here is a possible answer. (Note that the initialization of the instance variable, value, to zero is not really necessary, since it would be initialized to zero anyway if no explicit initialization were provided.)

```
/**
 * An object of this class represents a counter that
counts up from zero.
 */
public class Counter {

   private int value = 0;  // Current value of the
counter.
```

```
   /**
    * Add one to the value of the counter.
    */
   public void increment() {
      value++;
   }

   /**
    * Returns the current value of the counter.
    */
   public int getValue() {
      return value;
   }

} // end of class Counter
```

Question 14:

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.");
```

Answer:

The variable `headCount` is a variable of type `Counter`, so the only thing that you can do with it is call the instance methods `headCount.increment()` and `headCount.getValue()`. Call `headCount.increment()` to add one to the counter. Call `headCount.getValue()` to discover the current value of the counter. Note that you can't get at the value of the counter directly, since the variable that holds the value is a `private` instance variable in the `Counter` object. Similarly for `tailCount`. Here is the program with calls to these instance methods filled in:

```
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.
       headCount.increment() ;   // Count a "head", using
headCount
   else
       tailCount.increment() ;   // Count a "tail", using
tailCount
}
System.out.println(("There were " + headCount.getValue() +
" heads.");
System.out.println(("There were " + tailCount.getValue() +
" tails.");
```

Question 15:

Explain why it can **never** make sense to test "`if (obj.equals(null))`".

Answer:

If the value of `obj` is **not** null, then the test correctly returns `false`. However, if the value of `obj` **is** null, then there is no such thing as `obj.equals`, and the attempt to evaluate `obj.equals(null)` will cause a *NullPointerException* that will crash the program! The correct way to test if the value of `obj` is null is "`if (obj == null)`".

Answers for Quiz on Chapter 6

Question 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.

Answer:

An event is anything that can occur asynchronously, not under the control of the program, to which the program might want to respond. GUI programs are said to be "event-driven" because for the most part, such programs simply wait for events and respond to them when they occur. In many (but not all) cases, an event is the result of a user action, such as when the user clicks the mouse button, types a character, or clicks a button. The program might respond to a mouse-click on a canvas by drawing a shape, to a typed character by adding the character to an input box, or to a click on a button by clearing a drawing. More generally, a programmer can set up any desired response to an event by writing an event-handling routine for that event.

Question 2:

A central concept in JavaFX is the *scene graph*? What is a scene graph?

Answer:

A scene graph is a hierarchical data structure that represents the contents of a window. The scene graph is made up of "nodes" that represent visible components in the window, such as buttons and text-input boxes. Some of the nodes are "containers," which can contain other nodes. The GUI in a window is created by building a scene graph for window and setting the "root," or top, node of the scene graph to be the root of the window's scene.

Question 3:

JavaFX has standard classes called *Canvas* and *GraphicsContext*. Explain the purpose of these classes.

Answer:

A *Canvas* is a scene graph node that represents a drawing surface where a program can draw pictures. A canvas has an associated *GraphicsContext* that is used to do the drawing. The graphics context has instance methods for drawing shapes, such as lines and rectangles, on the associated canvas. It can also draw text and images. The graphics context also has properties that affect the appearance of the things it draws, such as the font that is used for text and the colors that are used for stroking and filling shapes.

Question 4:

Suppose that `canvas` is a variable of type *Canvas*. Write a code segment that will draw a green square centered inside a red circle on the canvas, as illustrated.

Answer:

Note: The size of the square and circle are not specified in the problem, so any size would be acceptable, as long as the square is in the exact middle of the circle. My solution draws the shapes in a 100-by-100 pixel canvas, leaving a 10-pixel space around the edges. It does not draw the white background, only the shapes.

To draw on the canvas, you need to use the canvas's *GraphicsContext*. Since it's not given in the problem, you have to get the graphics context by calling `canvas.getGraphicsContext2D()`. If `g` is the graphics context, then the methods for drawing the interior of circle and square are `g.fillOval(x,y,w,h)` and `g.fillRect(x,y,w,h)`. (There are no special methods for drawing circles and squares; you have to draw ovals and rectangles, using the same value for the width and for the height.) The graphics context's `setFill()` has to be called before filling each shape, to set the color for the shape.

```
GraphicsContext g = canvas.getGraphicsContext2D();
g.setFill(Color.RED);      // draw the shapes
g.fillOval(10,10,80,80);
g.setFill(Color.GREEN);
g.fillRect(30,30,40,40);
```

Question 5:

Draw the picture that will be produced by the following `for` loop, where `g` is a variable of type *GraphicsContext*:

```
for (int i=10; i <= 210; i = i + 50) {
   for (int j = 10; j <= 210; j = j + 50) {
      g.strokeLine(i,10,j,60);
   }
}
```

Answer:

The outer loop is executed for values of `i` equal to 10, 60, 110, 160, and 210. For **each** of these values, the inner loop is executed for `j` equal to 10, 60, 110, 160, and 210. The `strokeLine` command is therefore executed 25 times —and so, 25 different lines are drawn. These lines connect the five points (10,10), (60,10), (110,10), (160,10), and (210,10) to the five points (10,60), (60,60), (110,60), (160,60), and (210,60) in all possible pairings. Here is the picture:

Question 6:

JavaFX has a standard class called *MouseEvent*. What is the purpose of this class? What does an object of type *MouseEvent* do?

Answer:

When an event occurs, the system packages information about the event into an object. That object is passed as a parameter to the event-handling routine. Different types of events are represented by different classes of objects. An object of type *MouseEvent* represents an event that was generated because the user moved the mouse or manipulated its buttons. It contains information about the location of the mouse cursor and any modifier keys that the user is holding down. This information can be obtained by calling the instance methods of the object. For example, if `evt` is a *MouseEvent* object, then `evt.getX()` is the `x`-coordinate of the mouse cursor, and `evt.isShiftDown()` is a boolean value that tells you whether the user was holding down the Shift key.

Question 7:

Explain the difference between a `KeyPressed` event and a `KeyTyped` event.

Answer:

`KeyPressed` and `KeyTyped` events are generated when the user uses the keyboard, and both kinds of event are represented by event objects of type *KeyEvent*. But a `KeyPressed` event occurs each time the user presses a key, and a `KeyTyped` event occurs each time the user's typing generates a

character. It might take several key presses (and maybe key releases) to type a single character.

In more detail: A `KeyPressed` event is generated every time the user depresses one of the keys on a keyboard, even if doing so does not type a character. For example, pressing the shift key generates a `KeyPressed` event, even though the shift key, by itself, does not type anything. The corresponding *KeyEvent* object contains a code (such as `KeyCode.SHIFT` for the shift key) that tells which key was pressed. As the user types, a `KeyTyped` event is generated each time the user's typing generates a character. The `KeyTyped` event is in addition to any `KeyPressed` and `KeyReleased` events that were generated. The *KeyEvent* object for a `KeyTyped` event has a property of type *String* that contains the character that was typed.

Question 8:

Explain how *AnimationTimers* are used to do animation.

Answer:

Displaying an animation requires showing a sequence of frames. The frames are shown one after the other, with a short delay between each frame and the next. An *AnimationTimer* has an instance method, `handle()`, that is called repeatedly while the animation is running. That method can update any state variables that need to change between frames, and it can draw a picture for the current frame. To use an *AnimationTimer* you need to create a subclass of *AnimationTimer* and provide a definition for the `handle()` method. You need to create a variable, say `timer`, from that subclass, and call `timer.start()` to start the animation running. (You can call `timer.stop()` to stop the animation.)

Question 9:

What is a *CheckBox* and how is it used? How does it differ from a *RadioButton*?

Answer:

A *CheckBox* is a component that has two possible states, "selected" and "not selected". The user can change the state by clicking on the *CheckBox*. If `box` is a variable of type *CheckBox*, then a program can set the state of the box to "selected" by calling `box.setSelected(true)` and can deselect the box by calling `box.setSelected(false)`. The current state can be determined by calling `box.isSelected()`, which is a boolean-valued function. A *CheckBox* generates an event of type `ActionEvent` when the user changes its state. A program can listen for these events if it wants to take some action at the time the state changes. Often, however, it's enough for a program simply to look at the state of the *CheckBox* when it needs it.

A *RadioButton* is very similar and can be used in the same way. The difference is that a *RadioButton* can be added to a *ToggleGroup*. The toggle group will ensure that only one radio button in the group can be selected at a given time. When the user selects a radio button in the group, the previously selected button is automatically deselected.

Question 10:

What is meant by *layout* of GUI components? Explain briefly how layout is done in JavaFX.

Answer:

Layout means setting the sizes and the locations of the visible components in a window. In JavaFX, most components are in "containers", such as *VBox* or *BorderPane*, and it is the responsibility of the container to lay out the components that it contains. The programmer simply adds the components to the container (when building the scene graph). Different types of container

implement different policies for laying other their content. Some containers leave some or all of the layout decisions to the programmer, and it is also possible for the program to take over the layout of a component completely by making that component "unmanaged."

Question 11:

How does a *BorderPane* arrange the components that it contains?

Answer:

A *BorderPane* can contain up to five components. One component occupies the center of the pane while the other components, if present, are placed above, below, to the left and to the right of the center. (The position of a component, `c`, is specified when it is added to the pane, using one of the commands `pane.setCenter(c)`, `pane.setTop(c)`, `pane.setBottopm(c)`, `pane.setLeft(c)`, or `pane.setRight(c)`.)

The top and bottom components will be given their preferred heights; their widths will be set to fill the entire width of the pane (if that width is not greater than the component's maximum width). The left and right components will be shown at their preferred widths, and their height will be set to fill the height of the pane, minus the space occupied by the top and bottom components. The remaining space in the pane is used for the center component.

Question 12:

How is the *preferred size* of a component set, and how is it used?

Answer:

Standard components such as *Button* and *Label* are responsible for computing their own preferred size. A label, for example, has a preferred size that is just large enough to show the text and/or graphic on the label. The preferred size of a container is also usually computed by the container, based on the preferred sizes of the components that it contains. But is also possible for a program to set the preferred sizes of most components by calling the methods `comp.setPrefWidth(w)` and `comp.setPrefHeight(h)`, and those values will be used instead of a computed size. (There is also a method `comp.setPrefSize(w,h).`)

A container uses the preferred sizes of all the components in the container when it decides how to lay out those components. It also uses the components' preferred sizes if it needs to compute its own preferred size.

(Note that the preferred size is only preferred; it is not necessarily the size at which the component will appear on the screen. The actual size can depend on how container it is in does layout, the position of the component in the layout, the size of the container, and the preferred sizes of other components in the container.)

Answers for Quiz on Chapter 7

Question 1:

What is meant by the *basetype* of an array?

Answer:

The base type of an array refers to the type of the items that can be stored in that array. For example, the base type of the array of type `Color[]` is *Color*.

Question 2:

What is the purpose of the following variable-arity method? What are the values of `same(1,2,3)`, `same(17,17,17,17)`, and `same(2)`? Why?

```
static double same( int... value ) {
    for (int i = 1; i < value.length; i++) {
        if ( value[i-1] != value[i] )
            return false;
    }
    return true;
}
```

Answer:

This function can be called with any number of integer parameters. It tests whether all of the parameter values are equal. The value of `same(1,2,3)` is `false`. The function begins by comparing 1 and 2; since they are not equal, it immediately returns `false`. The value of `same(17,17,17,17)` is `true`. The function compares `value[0]` with `value[1]`, `value[1]` with `value[2]`, and `value[2]` with `value[3]`. All of these values are 17, so the routine goes through the `for` loop without returning `false`, and the "`return true`"

statement after the end of the loop is executed. For `same(2)`, `value.length` is 1, and the body of the `loop` is never executed; the return value is `true`.

Question 3:

What does it mean to sort an array?

Answer:

To sort an array means to rearrange the items in the array so that they are in increasing or decreasing order (according to some criterion).

(Note: To be more accurate, we should say "non-decreasing" instead of "increasing" and "non-increasing" instead of "decreasing." An array can contain duplicate values, and in that case, its elements can't be be put into strictly increasing or strictly decreasing order.)

Question 4:

What is the main advantage of binary search over linear search? What is the main disadvantage?

Answer:

The advantage of binary search is that it is much faster. On a list of one million items, linear search would take an average of five hundred thousand steps to find an item, whereas binary search would take only 20 steps. The disadvantage is that binary search only works on a sorted list, so some extra work must be done to sort the list or to keep the list in sorted order as it is created.

Question 5:

What is meant by a *dynamic array?* What is the advantage of a dynamic array over a regular array?

Answer:

A dynamic array is like an array in that it is a data structure that stores a sequence of items, all of the same type, in numbered locations. It is different from an array in that there is no preset upper limit on the number of items that it can contain. This is an advantage in situations where a reasonable value for the size of the array is not known at the time it is created.

Question 6:

What does it mean to say that *ArrayList* is a parameterized type?

Answer:

A class name is a type, but as a parameterized type, *ArrayList* does not define just one type. Instead, for every object type *T*, there is a type `ArrayList<T>`. *T* is a "type parameter." An object of type `ArrayList<T>` is a list that can only hold objects of type *T*. We can have `ArrayList<String>` to represent lists of *Strings*, `ArrayList<Integer>` to represent lists of *Integers*, and so one. The fact that *ArrayList* is a parameterized type means that we don't have to create a new class for each type of list.

Question 7:

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.

Answer:

Strings can be compared for lexicographic order using the `compareTo` instance method in the *String* class. We can find the smallest string as follows:

```
String smallest = strlst.get(0);
for (int i = 1; i < strlist.size(); i++) {
   String nextString = strlst.get(i);
   if ( nextString.compareTo(smallest) < 0 )
       smallest = nextString;
}
System.out.println("The smallest string lexicographically
is " + smallest);
```

Alternatively, a for-each loop could be used:

```
String smallest = strlst.get(0);
for (String item : strlst) {
   if ( item.compareTo(smallest) < 0 )
       smallest = item;
}
System.out.println("The smallest string lexicographically
is " + smallest)
```

Note that in this version, the first item in the list is compared to itself as the first step in the loop. There is no easy way to leave that item out of the for-each loop.

Question 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();
}
```

Answer:

The output consists of six lines, with each line containing six characters. In the first line, `i` is 0, so all the characters are *'s. In the last line, i is 5, so all the characters are *'s. In each of the four lines in the middle, one of the characters is a * and the rest are periods. The output is

```
******
.*....
..*...
...*..
....*.
******
```

It might help to look at the array items that are printed on each line. Note that `pic[row][col]` is '*' if `row` is 0 or if `row` is 5 or if `row` and `col` are equal. The items with value '*' are shown in red italic.

```
pic[0][0] pic[0][1] pic[0][2] pic[0][3] pic[0][4] pic[0]
[5]
pic[1][0] pic[1][1] pic[1][2] pic[1][3] pic[1][4] pic[1]
[5]
pic[2][0] pic[2][1] pic[2][2] pic[2][3] pic[2][4] pic[2]
[5]
pic[3][0] pic[3][1] pic[3][2] pic[3][3] pic[3][4] pic[3]
[5]
pic[4][0] pic[4][1] pic[4][2] pic[4][3] pic[4][4] pic[4]
[5]
pic[5][0] pic[5][1] pic[5][2] pic[5][3] pic[5][4] pic[5]
[5]
```

Question 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.

Answer:

One possible answer is:

```
public static int getMax(int[] list) {

   int max = list[0];  // This is the largest item seen so
far.

   for (int i = 1; i < list.length; i++) {
         // Look at each item in the array.  If the item
is
         // bigger than max, then set max equal to the
item.
       if (list[i] > max)
          max = list[i];
   }

   // At this point, max is the largest item in the whole
array.

   return max;

} // end getMax
```

(Note that this method throws an exception if the parameter `list` is null or if it is an array of length 0. The exception is thrown by the line "`int max = list[0];`". The reference to `list[0]` causes a *`NullPointerException`* if `list` is null and an *ArrayIndexOutOfBoundsException* if the array has length zero.)

Question 10:

Suppose that temperature measurements were made on each day of 2021 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.

Answer:

A pseudocode outline of the answer is

```
For each city {
   Add up all the temperatures for that city
   Divide the total by 365 and print the answer
}
```

Adding up all the temperatures for a given city itself requires a for loop, so the code segment looks like this:

```
for (int city = 0; city < 100; city++) {
    int total = 0;  // total of temperatures for this city
    for (int day = 0; day < 365; day++)
       total = total + temps[city][day];
    double avg = total / 365.0;  // average temp for this
city
    System.out.println("Average temp for city number "
             + city + " is " + avg);
}
```

Question 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. Write two versions of the code segment, one that uses a regular `for` loop, and one that uses a for-each loop.

Answer:

(The data for the i-th employee is stored in an object that can be referred to as `employeeData[i]`. The four pieces of data about that employee are members of this object and can be referred to as:

- `employeeData[i].firstName`
- `employeeData[i].lastName`
- `employeeData[i].hourlyWage`
- `employeeData[i].yearsWithCompany`

The code segment uses a for loop to consider each employee in the array.)

```
for (int i=0; i < 100; i++) {
    if ( employeeData[i].yearsWithCompany >= 20 )
        System.out.println(employeeData[i].firstName + " "
+
                        employeeData[i].lastName + ": " +
                        employeeData[i].hourlyWage);
}
```

Using a for-each loop:

```
for ( Employee emp : employeeData ) {
    if ( emp.yearsWithCompany >= 20 )
        System.out.println(emp.firstName + " " +
                        emp.lastName + ": " +
                        emp.hourlyWage);
}
```

Question 12:

Convert the *Employee* class from the previous question into a record class. What changes would then need to be made to the previous question's solution?

Answer:

It is easy to make the conversion, remembering that the instance variables of a record class are listed in parentheses after the class name. The class body will be empty:

```
record Employee(
   String lastName,
   String firstName,
   double hourlyWage,
   int yearsWithCompany
) { }
```

(*Employee* objects would than have to be created using the canonical constructor, and it would not be possible to change the value of any instance variable after the object has been created. However, none of that is relevant to the solution of the previous question.)

Since the fields of a record are private, they are accessed by calling methods instead of directly. For example, `emp.firstName` would become `emp.firstName()`. The extra parentheses are the only changes that are relevant to the question. So the two loops from the previous problem would become:

```
for (int i=0; i < 100; i++) {
    if ( employeeData[i].yearsWithCompany >= 20 )
        System.out.println(employeeData[i].firstName() + "
" +
                       employeeData[i].lastName() + ": " +
                       employeeData[i].hourlyWage() );
}
```

and

```
for ( Employee emp : employeeData ) {
    if ( emp.yearsWithCompany >= 20 )
```

```
        System.out.println(emp.firstName() + " " +
                      emp.lastName() + ": " +
                      emp.hourlyWage() );
}
```

(Note however that if the *Employee* class is a nested class in the same class that contains the `for` loops, then the change is not necessary, since a class has direct access to the private variables in a nested class that it contains.)

Question 13:

Write a record class to represent dates, where a *Date* object contains three integer fields giving the month, day, and year of the date. The canonical constructor should throw an exception if the values for the month and day are not legal. Also include a `toString()` method that prints a date in a form such as "5/27/2022".

Answer:

My solution assumes that months are numbered 1 through 12, and days are numbered 1 through 31. Since different months have different lengths, it's actually non-trivial to check for illegal values, especially taking into account the rule for leap years. But here is my solution:

```
public record Date( int month, int day, int year ) {

    public Date {  // the canonical constructor
        if ( month < 1 || month > 12 )
            throw new IllegalArgumentException("Illegal
month number.");
        int maxday; // last day of the month
        switch (month) {
            case 4, 6, 9, 11 -> maxday = 30;  // April,
June, September, October
            case 2 -> {  // February
                boolean isLeapYear = (year % 400 == 0)
                           || ((year % 4 == 0) && (year %
100 != 0));
                maxday = isLeapYear ? 29 : 28;
            }
```

```
            default -> maxday = 31;  // All other months
        }
        if ( day < 1 || day > maxday)
            throw new IllegalArgumentException("Illegal
day number.");
    }

    public String toString() {
        return String.format("%d/%d/%d", month, day,
year);
    }

}
```

Question 14:

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 double[] sums( double[][] data ) {
    double[] answers = new double[ data.length ];
    for (int i = 0; i < data.length; i++) {
        double sum = 0;
        for (int j = 0; j < data[i].length; i++)
            sum = sum + data[i][j];
        answers[i] = sum;
    }
    return answers;
}
```

Answer:

The purpose of this subroutine is to find the sum of the numbers in each row of the parameter array `data`. The answer is returned as a one-dimensional array of double. The value of `answers[i]` is the sum of the numbers in row number `i` of `data`.

(In the first two lines of the method, `data.length` is the number of rows in `data`. In the nested `for` loop, `data[i].length` is the number of values in the `i`-th row.)

Answers for Quiz on Chapter 8

Question 1:

Why do programming languages require that variables be declared before they are used? What does this have to do with correctness and robustness?

Answer:

It's a little inconvenient to have to declare every variable before it is used, but it's much safer. If the compiler would accept undeclared variables, then it would also accept misspelled names and treat them as valid variables. This can easily lead to incorrect programs. When variables must be declared, the unintentional creation of a variable is simply impossible, and a whole class of possible bugs is avoided.

Question 2:

What is a *precondition*? Give an example.

Answer:

A precondition is a condition that has to hold at a given point in the execution of a program, if the execution of the program is to continue correctly. For example, the statement "`x = A[i];`" has two preconditions: that `A` is not `null` and that `0 <= i < A.length`. If either of these preconditions is violated, then the execution of the statement will generate an error.

Also, a precondition of a subroutine is a condition that has to be true when the subroutine is called in order for the subroutine to work correctly.

Question 3:

Explain how preconditions can be used as an aid in writing correct programs.

Answer:

Suppose that a programmer recognizes a precondition at some point in a program. This is a signal to the programmer that an error might occur if the precondition is not met. In order to have a correct and robust program, the programmer must deal with the possible error. There are several approaches that the programmer can take. One approach is to use an `if` statement to test whether the precondition is satisfied. If not, the programmer can take some other action such as printing an error message and terminating the program. Another approach is to use a `try` statement to catch and respond to the error. This is really just a cleaner way of accomplishing the same thing as the first approach. The best approach, when it is possible, is to ensure that the precondition is satisfied as a result of what has already been done in the program. For example, if the precondition is that `x >= 0`, and the preceding statement is "`x = Math.abs(y);`", then we know that the precondition is satisfied, since the absolute value of any number is greater than or equal to zero.

Question 4:

Find a useful loop invariant for the `while` loop in the binary search algorithm (Subsection 7.5.1).

Answer:

The `binarySearch()` method searches for an integer `N` in a sorted array of integers `A[]`. The loop invariant is implicit in the discussion of the method in Subsection 7.5.1: "If `N` is in the array, then `N` is in at one of the indices in the range `lowestPossibleLoc` and `highestPossibleLoc`."

This statement is true before the loop is started, since at that point, the range of indices from `lowestPossibleLoc` to `highestPossibleLoc` includes the entire array. In the loop, when `A[mid]` is compared to `N`, the fact that `A` is sorted means that half of the range of possible indices can be discarded. After the value of `lowestPossibleLoc` or `highestPossibleLoc` is modified to take that into account, the loop invariant remains true.

If at any point `N` is found, the loop ends and the method returns the correct value. If `N` is never found, the loop will end when `lowestPossibleLoc` is larger than `highestPossibleLoc`. At that point, the truth of the loop invariant implies that `N` is not in the array.

Question 5:

Java has a predefined class called *Throwable*. What does this class represent? Why does it exist?

Answer:

The class *Throwable* represents all possible objects that can be thrown by a `throw` statement and caught by a `catch` clause in a `try..catch` statement. That is, the thrown object must belong to the class *Throwable* or to one of its (many) subclasses such as *Exception* and *RuntimeException*. The object carries information about an exception from the point where the exception occurs to the point where it is caught and handled.

Question 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*.

Answer:

The problem of large values in a `3N+1` sequence was discussed in Section 8.1. In that section, it is pointed out that the test "`if (N > 2147483646/3)`" can be used to test whether the value of `N` has become too large. This test is used in the following method.

```
/** Print the 3N+1 sequence starting from N.  If N
 * is not greater than 0 or if the value of N exceeds
 * the maximum legal value for ints, than an
 * exception will be thrown.
 */
static void printThreeNSequence(int N) {
   if (N < 1) {
      throw new IllegalArgumentException(
                  "Starting value for 3N+1 sequence must
be > 0.");
   }
   System.out.println("3N+1 sequence starting from " + N +
" is: ");
   System.out.println(N);
   while (N > 1) {
      if (N % 2 == 0) {  // N is even.  Divide by 2.
          N = N / 2;
      }
      else {  // N is odd.  Multiply by 3 and add 1.
          if (N > 2147483646/3) {
             throw new ArithmeticException("Value has
exceeded the largest int.");
          }
          N = 3 * N + 1;
      }
      System.out.println(N);
   }
}
```

(Note that it would be possible to declare that this routine can throw exceptions by adding a "throws" clause to the heading:

```
static void printThreeNSequence(int N)
           throws IllegalArgumentException,
ArithmeticException {
```

However, this is not required since `IllegalArgumentException`s and `ArithmeticException`s are not checked exceptions.)

Question 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?

Answer:

We can replace the `if` statements that check for errors with assert statements that give the same error messages:

```
/** Print the 3N+1 sequence starting from N.
  * Precondition:  N > 0 and the 3N+1 sequence for N does
not contain
  * any numbers that are too big to be represented as 32-
bit ints.
  */
static void printThreeNSequence(int N) {

   assert  N > 0 : "Starting value for 3N+1 sequence must
be > 0.";

   System.out.println("3N+1 sequence starting from " + N +
" is: ");

   System.out.println(N);
   while (N > 1) {
      if (N % 2 == 0) {  // N is even.  Divide by 2.
          N = N / 2;
      }
      else {  // N is odd.  Multiply by 3 and add 1.
          assert  N <= 2147483646/3 : "Value has exceeded
the largest int.";
          N = 3 * N + 1;
      }
      System.out.println(N);
   }
```

```
}
```

The first version of the method will always check for errors when the program is run. The second version, on the other hand, does not actually do any error checking when the program is run in the ordinary way. In order for `assert` statements to be executed, the program must be run with assertions enabled. The `assert` statements are really there only to do error-checking during debugging and testing. (In this particular case, I would say that an exception should definitely be thrown when `N` exceeds the maximum legal value, but that it's reasonable to use an `assert` to check whether `N > 0`.)

Question 8:

Some classes of exceptions are *checked exceptions* that require *mandatory exception handling*. Explain what this means.

Answer:

Subclasses of the class *Exception* which are not subclasses of *RuntimeException* are checked exceptions. This has two consequences: First, if a method can throw such an exception, then it must declare this fact by adding a `throws` clause to the method heading. Second, if a routine includes any code that can generate such an exception, then the routine must deal with the exception. It can do this by including the code in a `try` statement that has a `catch` clause to handle the exception. Or it can add a `throws` clause to the method definition to declare that calling the method might throw the exception.

Question 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.

Answer:

```
try {
   processData();
}
catch (IOException e) {
   System.out.println("An IOException occurred while
processing the data.");
}
```

Question 10:

Why should a subroutine throw an exception when it encounters an error? Why not just terminate the program?

Answer:

Terminating the program is too drastic, and this tactic certainly doesn't lead to robust programs! It's likely that the subroutine doesn't know what to do with the error, but that doesn't mean that it should abort the whole program. When the subroutine throws an exception, the subroutine is terminated, but the program that called the subroutine still has a chance to catch the exception and handle it. In effect, the subroutine is saying "Alright, I'm giving up. Let's hope someone else can deal with the problem."

Question 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?

Answer:

In the absence of other information, the second algorithm, with run time $\Theta(n*\log(n))$, is the better choice, since $n*\log(n)$ is much smaller than n^2, for most values of n. However, it's not completely certain that the second algorithm is the better choice in a particular case. First of all, although the $n*\log(n)$ algorithm is certainly better than the n^2 algorithm for large enough values of n, that is not necessarily true for $n = 100$. Second, there is the issue of "average-case" run time. Even if the $n*\log(n)$ algorithm is better for most inputs of size 100, it might not be better for all such inputs.

Question 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 $O(f(n))$ 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];
}
```

Answer:

The run time of this algorithm is $\Theta(n)$. There are several things in the code that are evaluated n times: the test "i < A.length", the increment "i++", and the test in the if statement. The initialization is done once, and nothing is executed more than n times. It follows that both the worst-case and the average case run times are $\Theta(n)$.

Answers for Quiz on Chapter 9

Question 1:

Explain what is meant by a *recursive* subroutine.

Answer:

A recursive subroutine is simply one that calls itself either directly or through a chain of calls involving other subroutines.

Question 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.print("]");
    }
}
```

Show the output that would be produced by the subroutine calls `printStuff(0)`, `printStuff(1)`, `printStuff(2)`, and `printStuff(3)`.

Answer:

The outputs are:

```
printStuff(0) outputs:   *
printStuff(1) outputs:   [*,*]
printStuff(2) outputs:   [[*,*],[*,*]]
printStuff(3) outputs:   [[[*,*],[*,*]],[[*,*],[*,*]]]
```

(Explanation: For `printStuff(0)`, the value of the parameter is 0, so the first clause of the `if` is executed, and the output is just *. For `printStuff(1)`, the else clause is executed. This else clause contains two recursive calls to `printStuff(level-1)`. Since `level` is 1, `level-1` is 0, so each call to `printStuff(0)` outputs a *. The overall output from `printStuff(1)` is [*,*]. In a similar way, `printStuff(2)` includes two recursive calls to `printStuff(1)`. Each call to `printStuff(1)` outputs [*,*]. And `printStuff(2)` just takes two copies of this and puts them between [and] separated by a comma: [[*,*],[*,*]]. Finally, the output from `printStuff(3)` outputs two copies of [[*,*],[*,*]] separated by a comma and enclosed between brackets. Once you recognize the pattern, you can do `printStuff(N)` for any `N` without trying to follow the execution of the subroutine in detail.)

Question 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.

Answer:

I'll give both a non-recursive solution and a recursive solution. For a linked list, the recursion is not really necessary, but it does nicely reflect the recursive definition of `ListNode`

```
static int countZeros( ListNode head ) {
   int count;         // The number of zeros in the list.
   ListNode runner;   // For running along the list.
   count = 0;
   runner = head;     // Start at the beginning of the
list.
   while (runner != null) {
      if ( runner.item == 0)
         count++;  // Count the zero found in the current
node.
      runner = runner.next;  // Advance to the next node.
   }
   return count;
}

static int countZerosRecursively( ListNode head ) {
   if ( head == null) {
          // An empty list does not contain any zeros.
       return 0;
   }
   else {
       int count = countZerosRecursively( head.next );  //
Count zeros in tail.
       if ( head.item == 0 )
           count++;  // Add 1 to account for the zero in
the head node.
       return count;
   }
}
```

Question 4:

Let *ListNode* be defined as in the previous problem. Suppose that `head` is a variable of type *ListNode* that points to the first node in a linked list. Write a code segment that will add the number 42 in a new node at the **end** of the list. Assume that the list is not empty. (There is no "tail pointer" for the list.)

Answer:

```
ListNode tail;  // Since no tail pointer is given, we need
to make one
tail = head;  // We are assuming that this is not null.
```

```
while (tail.next != null) {
    tail = tail.next;
}
// At this point, tail points to the last node in the list
ListNode node42;
node42 = new ListNode();  // create a new node
node42.item = 42;    // the item in the new node is 42
tail.next = node42;  // attach the new node to the list
```

(Explanation: To add a list node at the end of the list, we need a pointer to the last node in the list. Since no such pointer is given, we have to make one. Start with a pointer that is equal to `head`, and move it down the list until it points to the last node in the list. We know that `tail` is pointing to the last node in the list if `tail.next` is `null`. Use a `while` loop to move `tail` down the list until that is true. Then we can make a new node and attach it to the list by setting `tail.next` to point to it. The value of `node42.next` is `null`, and that `null` marks the new end of the list.)

Question 5:

List nodes can be used to build linked data structures that do not have the form of linked lists. Consider the list node class shown on the left and the code shown on the right:

```
class ListNode {
    int item;
    ListNode next;
    Listnode(int i) {
        item = i;
        next = null;
    }
}
```

```
ListNode one = new ListNode(10);
ListNode two = new ListNode(20);
ListNode three = new ListNode(30);
ListNode four = new ListNode(40);

one.next = two;
two.next = three;
three.next = four;
four.next = two;
```

Draw the data structure that is constructed by the code. What happens if you try to print the items in the data structure using the usual code for traversing a linked list:

```
ListNode runner = one;
while (runner != null) {
    System.out.println(runner.item);
    runner = runner.next();
}
```

Answer:

Node one links to node two, node two links to node three, and node three links to node four. If `four.next` were `null`, the result would be a normal, four-node linked list. However, node four links back to node two:

When the while loop is executed, runner will never become null. After the values in the four nodes (10, 20, 30, 40) are printed, the assignment `runner = runner.next` will set the runner to point to node two again, so the next number printed is 20. The program goes into an infinite loop in which the numbers 20, 30, 40 are printed over and over forever.

Question 6:

What are the three operations on a *stack?*

Answer:

The three stack operations are *push*, *pop,* and *isEmpty.* The definitions of these operations are: `push(item)` adds the specified item to the top of the stack; `pop()` removes the top item of the stack and returns it; and `isEmpty()` is a boolean-valued function that returns true if there are no items on the stack.

Question 7:

What is the basic difference between a stack and a queue?

Answer:

In a stack, items are added to the stack and removed from the stack on the same end (called the "top" of the stack). In a queue, items are added at one end (the "back") and removed at the other end (the "front"). Because of this difference, a queue is a FIFO structure (items are removed in the same order in which they were added), and a stack is a LIFO structure (the item that is popped from a stack is the one that was added most recently).

Question 8:

What is an *activation record*? What role does a stack of activation records play in a computer?

Answer:

When a subroutine is called, an activation record is created to hold the information that is needed for the execution of the subroutine, such as the values of the parameters and local variables. This activation record is stored on a stack of activation records. A stack is used since one subroutine can call another, which can then call a third, and so on. Because of this, many activation records can be in use at the same time. The data structure is a stack because an activation record has to continue to exist while all the subroutines that are called by the subroutine are executed. While they are being executed, the stack of activation records can grow and shrink as subroutines are called and return.

Question 9:

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.

Answer:

```
static int treeSum( TreeNode root ) {
       // Find the sum of all the nodes in the tree to
which root points.
    if ( root == null ) {
          // The sum of the nodes in an empty tree is
zero.
       return 0;
    }
    else {
          // Add the item in the root to the sum of the
          // items in the left subtree and the sum of the
          // items in the right subtree.
       int total = root.item;
       total += treeSum( root.left );
       total += treeSum( root.right );
       return total;
    }
 }
```

Question 10:

Let *TreeNode* be the same class as in the previous problem. Write a recursive subroutine that makes a copy of a binary tree. The subroutine has a parameter that points to the root of the tree that is to be copied. The return type is *TreeNode*, and the return value should be a pointer to the root of the

copy. The copy should consist of newly created nodes, and it should have exactly the same structure as the original tree.

Answer:

```
static TreeNode copyTree( TreeNode root ){
        // Make a copy of the tree that root points to.
    if (root == null) {
        return null;  // The copy of an empty tree is an
empty tree
    }
    else {
            // The tree is not empty.  We need to make a
new node
            // to be the root of the copy, and then we
need to copy
            // the left and right subtrees.
         TreeNode rootOfCopy = new TreeNode();
         rootOfCopy.item = root.item;
         rootOfCopy.left = copyTree( root.left );
         rootOfCopy.right = copyTree( root.right );
         return rootOfCopy;
    }
}
```

(The essential point here is that the subtrees of the original tree can be copied using recursive calls to `copyTree()`. For example, `copyTree(root.left)` will make a complete copy of the left subtree of the original tree, and that can become the left subtree of the new root node. This works even if `root.left` is `null`.)

Question 11:

What is a *postorder traversal* of a binary tree?

Answer:

In a traversal of a binary tree, all the nodes are processed in some way. (For example, they might be printed.) In a postorder traversal, the order of processing is defined by the rule: For each node, the nodes in the left subtree of that node are processed first. Then the nodes in the right subtree are processed. Finally, the node itself is processed. This rule is applied at all levels of the tree.

Question 12:

Suppose that a binary sort tree of integers is initially empty and that the following integers are inserted into the tree in the order shown:

```
5    7    1    3    4    2    6
```

Draw the binary sort tree that results. Then list the integers in the order that is produced by a post-order traversal of the tree.

Answer:

This picture shows the tree that results after each integer has been inserted, using the `treeInsert()` method from Subsection 9.4.2:

In a post-order traversal, the nodes in the left subtree are processed first, then the nodes in the right subtree, then the root node. For this tree, the nodes

are processed in the order:

```
2    4    3    1    6    7    5
```

(Of course, an in-order traversal would process the nodes in sorted order: 1, 2, 3, 4, 5, 6, 7.)

Question 13:

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.)

Answer:

Here are five possibilities (out of an infinite number of possibilities), with some explanation:

```
fred — A <multilist> can just be a word, such as "fred".

( )  — The [ ]... around <multilist> means that there can
be
          any number of nested <multilist>'s, including
zero.  If
          there are zero, then all that's left is the
empty
          parentheses.

( fred mary chicago )—A <multilist> consisting of three
                         <multilist>'s—"fred", "mary", and
                         "chicago"—inside parentheses

( ( able ) ( baker charlie ) )—A <multilist> containing
two
                                    <multilist>'s.
```

```
( ( a ( b ) ) ( c ( d e ) g ) ) —Even more nesting.
```

Question 14:

Explain what is meant by *parsing* a computer program.

Answer:

To parse a computer program means to determine its syntactic structure, that is, to figure out how it can be constructed using the rules of a grammar (such as a BNF grammar).

Answers for Quiz on Chapter 10

Question 1:

What is meant by *generic programming* and what is the alternative?

Answer:

Generic programming means writing data structures and subroutines that can be used for many different types of data. The alternative would be to write a new data structure or subroutine for each different type of data, even when they would all be essentially identical except for the type name. For example, a single generic sort routine can be used for sorting lists that contain data of any type. The alternative is one routine for sorting lists of integers, one for sorting lists of strings, one for storing arrays of real numbers, and so on.

Question 2:

Why can't you make an object of type *LinkedList<int>*? What should you do instead?

Answer:

LinkedList<int> is an attempt to use a generic class with a type parameter of type int, which is a primitive type. Generic programming in Java works only for objects, and not for the primitive types, so a type parameter of primitive type is not allowed. However, it is possible to use the wrapper class *Integer* in place of int. An object of type *LinkedList<Integer>* can be used almost as if it were actually a list of ints.

Question 3:

What is an *iterator* and why are iterators necessary for generic programming?

Answer:

One of the principle features of Java's generic programming framework is *Collections*. There are several types of collection (including *LinkedList*, *ArrayList*, *TreeSet*, and *HashSet*). In order to deal with all the different types of collection in a generic way, we need a generic way to access all the elements in a collection. An iterator makes this possible. An iterator is an object associated with a collection that makes it possible to traverse the collection (that is, to visit each of the items in the collection in turn). Code written using iterators will work for any type of collection. (Note, however, that explicit use of an iterator can often be avoided by using a for-each loop.)

Question 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.

Answer:

Using an iterator:

```
int sum = 0;
Iterator<Integer>  iter = integers.iterator();
while ( iter.hasNext() ) {
   sum += iter.next();
}
```

The statement "`sum += iter.next()`" relies on the automatic conversion from type *Integer* to type int. It could also be written "`sum += iter.next().intValue()`".

Using a for-each loop:

```
int sum = 0;
for ( int number : integers ) {    // ( Could also use
"Integer number : integers". )
   sum += number;
}
```

Note that either of these two solutions will fail if `integers` contains a `null` value.

Question 5:

Interfaces such as *List*, *Set*, and *Map* define *abstract data types*. Explain what this means.

Answer:

An abstract data type is defined by the operations that can be performed on it, not by the way the data is actually stored or by the way the operations are implemented. An interface such as *List* defines operations that can be performed, but says nothing about how they are to be implemented. In fact, there can be many different implementations. For example, both *LinkedList* and *ArrayList* implement the *List* interface. They are different "concrete" data types that implement the same abstract data type.

Question 6:

What is the fundamental property that distinguishes *Sets* from other types of *Collections*?

Answer:

A *Set* cannot contain duplicate elements. Adding an item to the set has no effect if that item is already in the set. (Note that exactly what it means to say

that two items are the same depends on the type of set. For *HashSet*, two items are tested for equality using the `equals()` method. For a *TreeSet*, the test for equality uses the `compareTo()` method from the *Comparable* interface or the `compare()` method from a *Comparator*.)

Question 7:

What is the essential difference in functionality between a *TreeMap* and a *HashMap*?

Answer:

The key/value pairs in a *TreeMap* are sorted so that the keys are in ascending order. (For this reason, it must be possible to compare the keys in a *TreeMap*, using a `compareTo()` method or `compare()`. Either the keys must implement the *Comparable* interface or a *Comparator* must be provided to do the comparison.)

Question 8:

Explain what is meant by a *hash code*.

Answer:

The hash code of an object is an integer that tells where that object should be stored in a hash table. A hash table is an array of linked lists. When an object is stored in a hash table, it is added to one of these linked lists. The object's hash code is the index of the position in the array where the object is stored. All objects with the same hash code go into the same linked list. In Java, every object `obj` has a method `obj.hashCode()` that is used to compute hash codes for the object. If the object is to be stored in a hash table of size `N`, then the hash code that is used for the object is

`Math.abs(obj.hashCode()) % N`.

Question 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;
   }
}
```

Also, rewrite the resulting *Date* class as a record class. (See Section 7.4.)

Answer:

The interface *Comparable<Date>* specifies the method `"public int compareTo(Date d)"`, which will be used to compare two objects of type *Date*. The `compareTo()` method must be added to the class, and the class must be declared to implement the interface. To compare two dates, first try comparing the years. If the years are equal, try comparing the months. If the months are also equal, compare the days.

```
class Date implements Comparable<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;
   }
   public int compareTo( Date otherDate ) {
           // Returns 1, 0, or -1 if this date is greater
than, equal to,
```

```
            // or less than otherDate, respectively.
      if (year < otherDate.year)
         return -1;
      else if (year > otherDate.year)
         return 1;
      else { // Years are equal; compare months.
         if (month < otherDate.month)
            return -1;
         else if (month > otherDate.month)
            return 1;
         else { // Years and months are equal; compare
days.
            if (day < otherDate.day)
               return -1;
            else if (day > otherDate.day)
               return 1;
            else
               return 0;
         }
      }
   }
}
```

When this class is rewritten as a record class, the class's instance variables are listed in parentheses after the class name, and a canonical constructor is automatically provided. However, the *Comparable* interface is added to the class in the same way. Record classes cannot extend other classes, but they can implement interfaces. Of course, this record class could use many other improvements...

```
record Date(int month, int day, int year)
                         implements Comparable<Date> {
   public int compareTo( Date otherDate ) {
            // Returns 1, 0, or -1 if this date is greater
than, equal to,
            // or less than otherDate, respectively.
      if (year < otherDate.year)
         return -1;
      else if (year > otherDate.year)
         return 1;
      else { // Years are equal; compare months.
         if (month < otherDate.month)
            return -1;
         else if (month > otherDate.month)
            return 1;
         else { // Years and months are equal; compare
```

```
days.
            if (day < otherDate.day)
               return -1;
            else if (day > otherDate.day)
               return 1;
            else
               return 0;
         }
      }
   }
}
```

Question 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, 2021.

Answer:

I will give two solutions. One of them simply looks up each date in December, 2021 in the map and prints the corresponding value, if there is one. The other iterates though a submap that contains all the entries for dates in that month.

```
A solution using the map's get() method:

      for (int day = 1; day <= 31; day++) {
           // Get the info for one day in December, 2021
         Date date = new Date(12,day,2021); // The key.
         String info = syllabus.get(date);  // Get the
value for that key.
                                            // (Can be
null if there is no
                                            // entry in
the map for this date.)
         if (info != null)
            System.out.println("December " + day + ": " +
info);
      }
```

```
A solution using a submap (harder, but more efficient):

      Date startDate = new Date(12,1,2021); // Starting
date for submap.
      Date endDate = new Date(1,1,2022);    // Ending date
for submap.
                                            // (Remember
that the end date
                                            // is not
included.)
      Map<Date,String> decemberData =
syllabus.subMap(startDate, endDate);
      for ( Map.Entry<Date,String> entry : decemberData )
{
         Date date = entry.getKey();
         String info = entry.getValue();
         System.out.println("December " + date.day + ": "
+ info);
      }

Using var to declare local variables, this could also be
written:

      Date startDate = new Date(12,1,2021); // Starting
date for submap.
      Date endDate = new Date(1,1,2022);    // Ending date
for submap.
                                            // (Remember
that the end date
                                            // is not
included.)
      var decemberData = syllabus.subMap(startDate,
endDate);
      for (var entry : decemberData ) {
         Date date = entry.getKey();
         String info = entry.getValue();
         System.out.println("December " + date.day + ": "
+ info);
      }
```

Question 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.

Answer:

```
public class Stack<T> {
   ArrayList<T> stack = new ArrayList<T>();
   public void push( T newItem ) {
      stack.add(newItem);
   }
   public T pop() {
      if ( isEmpty() ) {
          throw new IllegalStateException("Can't pop from
an empty stack");
      }
      int top = stack.size() - 1;  // location of top item
      return stack.remove(top);    // remove and return
top item
   }
   public boolean isEmpty() {
      return stack.size() == 0;
   }
}
```

Question 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.

Answer:

Since the method operates on *ArrayLists*, it can use indexed access with the `get(i)` and `set(i,item)` methods. These operations are efficient for array

lists. I also give a second version of the method that uses a list iterator and is efficient for any type of list.

```
public static <T> void replaceAll(ArrayList<T> list, T
oldItem, T newItem) {
   if (oldItem == null) {
      for (int i = 0; i < list.size(); i++) {
         if ( null == list.get(i) )
            list.set( i, newItem );
      }
   }
   else {
      for (int i = 0; i < list.size(); i++) {
         if ( oldItem.equals(list.get(i)) )
            list.set( i, newItem );
      }
   }
}


public static <T> void replaceAll(List<T> list, T oldItem,
T newItem) {
   ListIterator<T> iter = list.listIterator();
   while (iter.hasNext()) {
      T listItem = iter.next();
      if ( oldItem == null ) {
         if ( listItem == null )
            iter.set(newItem);
      }
      else {
         if ( oldItem.equals(listItem) )
            iter.set(newItem);
      }
   }
}
```

(Note, by the way, that a `replaceAll` method is already defined as a static method in class *Collections*.)

Question 13:

Suppose that `words` is an array of *Strings*. Explain what is done by the following code:

```
long n = Arrays.stream(words)
                .filter( w -> (w != null) )
                .map( w -> w.toLowerCase() )
                .distinct()
                .count();
```

Answer:

This code counts the number of different strings in the array, ignoring case. It starts with a stream containing all the strings from the array. The `filter()` operation only lets non-null values through. This makes sure that there won't be a `NullPointerExcetion` in the next step, when `w.toLowerCase()` is evaluated. The `map()` replaces each string in the stream with a lower case version, so that there will be no difference between upper and lower case versions of the same string. The `distinct() eliminates duplicates from the stream. And count() returns the number of strings in the resulting. stream, which now contains just one lower-case copy of every string that occurs in the array.`

Question 14:

Use the stream API to print all the even integers from 2 to 20. Start with `IntStream.range` and apply a `filter` operation.

Answer:

```
IntStream.range(2,20)
         .filter( n -> (n % 2 == 0) )
         .forEach( n -> System.out.println(n) );
```

(`IntStream.range()` can be used to create a stream containing all of the integers 2 to 20. The predicate "`n -> (n%2 == 0)`" is true just for even integers, so the stream that emerges from the `filter` operation contains the numbers 2, 4, ... 20. Finally, we can apply an operation to each of the numbers of the stream by using the `forEach` terminal operator. The operation `forEachOrdered` would also work, but since the stream is sequential,

`forEach` will print out the numbers in order. If we were using a parallel stream, it is possible that `forEach` could print the numbers out of order.)

Question 15:

Write a generic method `countIf(c,t)` with type parameter `<T>`, where the first parameter, `c`, is of type `Collection<T>`, and the second parameter, `p`, is of type `Predicate<T>`. The method should return the number of items in the collection for which the predicate is true. Give two versions, one using a loop and the other using the stream API.

Answer:

The first version uses a for-each loop, and uses the fact that a *Predicate* contains a boolean-valued function named `test()` for testing whether the predicate is true:

```
public static <T> int countIf( Collection<T> items,
Predicate<T> pred ) ) {
    int count = 0;
    for ( T x : items ) {
        if ( pred.test(x) )
            count++;
    }
    return count;
}
```

The second version uses the stream API, with a `filter()` operation that lets through only the items for which the predicate is true. Note that `filter()` takes a parameter that is a predicate. Usually, the parameter is given by a lambda expression, but in this case, the predicate is `pred`, so we need to pass `pred` as the paramter to `filter()`.

```
public static <T> int countIf( Collection<T> items,
Predicate<T> pred ) {
    long ct = items.parallelStream()
                   .filter(pred)
                   .count();
         // .count() returns a long, but I want countIf to
```

```
return an int
    return (int)ct;
}
```

Answers for Quiz on Chapter 11

Question 1:

In Java, input/output is done using I/O streams. I/O streams are an *abstraction.* Explain what this means and why it is important.

Answer:

An I/O stream represents a source from which data can be read or a destination to which data can be written. A stream is an abstraction because it represents the abstract idea of a source or destination of data, as opposed to specific, concrete sources and destinations such as a particular file or network connection. The I/O stream abstraction is important because it allows programmers to do input/output using the same methods for a wide variety of data sources and destinations. It hides the details of working with files, networks, and the screen and keyboard.

Question 2:

Java has two types of I/O stream: character streams and byte streams. Why? What is the difference between the two types of streams?

Answer:

Character streams are for working with data in human-readable format, that is, data expressed as sequences of characters. Byte streams are for data expressed in the machine-readable format that is used internally in the computer to represent the data while a program is running. It is very efficient for a computer to read and write data in machine format, since no translation of the data is necessary. However, if a person must deal directly with the data, then character streams should be used so that the data is presented in human-readable form.

Question 3:

What is a *file?* Why are files necessary?

Answer:

A file is a collection of data that has been given a name and stored on some permanent storage device such as a hard disk or USB memory stick. Files are necessary because data stored in the computer's RAM is lost whenever the computer is turned off. Data that is to be saved permanently must be stored in a file. (Furthermore, RAM is very expensive compared to space on a disk drive, so a computer's hard disk can typically store much more data than would fit in the computer's RAM, even if the computer were left turned on all the time.)

Question 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*?

Answer:

The *PrintWriter* class is being used as a "wrapper" for the *FileWriter* class. A *FileWriter* is a stream object that knows how to send individual characters to a file. By wrapping this in a *PrintWriter*, you get the ability to write other data types such as ints, doubles, and *Strings* to the file using the *PrintWriter's* `print()` and `println()` methods. Wrapping the `FileWriter` in a *PrintWriter* adds capabilities to the file output stream but still sends the data to the same destination.

Question 5:

The package `java.io` includes a class named *URL*. What does an object of type *URL* represent, and how is it used?

Answer:

A url is an address for a web page (or other information) on the Internet. For example, "https://math.hws.edu/javanotes/index.html" is a url that refers to the main page of the current edition of this on-line textbook. A *URL* object represents such an address. Once you have a *URL* object, you can call its `openConnection()` method to access the information at the web address that it represents.

Question 6:

What is the purpose of the *FileChooser* class?

Answer:

An object of type *FileChooser* represents a dialog box that can be used in a GUI program to let the user select a file that is to be used for input or for output.

Question 7:

Explain what is meant by the *client / server* model of network communication.

Answer:

In the client/server model, a server program runs on a computer somewhere on the Internet and "listens" for connection requests from client programs. The server makes some service available. A client program connects to the server to access that service. For example, a Web server has a collection of Web pages. A Web browser acts as a client for the Web server. It makes a connection to the server and sends a request for one of its pages. The server responds by transmitting a copy of the requested page back to the client.

Question 8:

What is a *socket*?

Answer:

A socket represents one endpoint of a network connection. A program uses a socket to communicate with another program over the network. Data written by a program to the socket at one end of the connection is transmitted to the socket on the other end of the connection, where it can be read by the program at that end.

Question 9:

What is a *ServerSocket* and how is it used?

Answer:

A *ServerSocket* is used by a server program to listen for connection requests from client programs. If `listener` refers to an object that belongs to Java's *ServerSocket* class, then calling the function `listener.accept()` will wait for a connection request and will return a *Socket* object that can be used to communicate with the client that made the request.

Question 10:

What is meant by an *element* in an XML document?

Answer:

An element consists of a tag (possibly containing attributes), a matching end-tag, and everything in between. There are also empty elements where an empty element consists of a single self-closing tag. An element can contain textual content and nested elements. For example, in the XML fragment

```
<testresult score="82"><name>Joe Smith</name>
<subject>Math</subject></testresult>
```

the entire fragment is an element, while `<name>Joe Smith</name>` and `<subject>Math</subject>` are nested elements inside the `<testresult>` element. "Joe Smith" and "Math" are examples of textual content, and `score` is an attribute with value `82`.

Question 11:

What is it about XML that makes it suitable for representing almost any type of data?

Answer:

XML is a syntax for building data representation languages. In XML, the names of tags and the structure of a document can be chosen to be whatever is most appropriate for the type of data that is being represented. The tag names can be chosen to give *meaningful* descriptions of the data. This contrasts with HTML documents, which can include only a certain fixed set of tags. (XML also has the advantage of being a widely accepted standard that is supported in just about every programming language.)

Question 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; read from the file using methods covered in this chapter.

Answer:

I will give three different solutions. The first two solutions use the standard classes *BufferedReader* and *Scanner* in the usual way. The third solution uses the stream API and the *BufferedReader* `lines()` method. All of these classes make it easy to read a line of text from an input stream.

For each program, I do everything in one big `try` statement. If anything goes wrong, an error message is printed in the `catch` clause of the `try` statement. For example, if the program is run with no command-line argument, an *IndexOutOfBoundsException* will be generated when the program refers to `args[0]`. If a file is specified, but it doesn't exist, then a *FileNotFoundException* will occur. (The exercise says to assume that the file has at least ten lines, but if there are fewer than ten lines in the file, the *Scanner* solution will crash, the first *BufferedReader* solution will just print a `null` for each missing line, and the second *BufferedReader* solution will stop after it has read all the lines that the file contains.)

```
import java.io.*;
import java.util.Scanner;
public class TenLinesWithScanner {

   public static void main(String[] args) {
      try {
         Scanner scanner = new Scanner( new File(args[0])
);
         for (int lineCt = 0; lineCt < 10; lineCt++) {
            String line = scanner.nextLine();
            System.out.println(line);
         }
```

```
      }
      catch (Exception e) {
         System.out.println("Error: " + e);
      }
   }

}  // end class TenLinesWithScanner

//-------------------------------------------------------------
---------

import java.io.*;
public class TenLinesWithBufferedReader {

   public static void main(String[] args) {
      try {
         BufferedReader in = new BufferedReader( new
FileReader(args[0]) );
         for (int lineCt = 0; lineCt < 10; lineCt++) {
            String line = in.readLine();
            System.out.println(line);
         }
      }
      catch (Exception e) {
         System.out.println("Error: " + e);
      }
   }

}  // end class TenLinesWithBufferedReader

//-------------------------------------------------------------
---------

import java.io.*;
public class TenLinesWithStreamAPI {

    public static void main(String[] args) {
            try {
                 BufferedReader in = new BufferedReader(
new FileReader(args[0]) );
                 in.lines()     // Make a Stream<String>
containing lines from file.
                   .limit(10)   // Truncate the string
after at most 10 items.
                   .forEach( s -> System.out.println(s) );
// Print each line.
            }
            catch (Exception e) {
```

```
                System.out.println("Error: " + e);
            }
    }
} // end class TenLinesWithStreamAPI
```

Note: I didn't close the files that were used in any of these solutions, although it would have been good form to do so. The easiest way to make sure that the file is closed is to use a `try..catch` statement with resources. (See the last few paragraphs of Subsection 8.3.2.) For example:

```
try ( BufferedReader in = new BufferedReader(new
FileReader(args[0])) ) {
   for (int lineCt = 0; lineCt < 10; lineCt++) {
      String line = in.readLine();
      System.out.println(line);
   }
}
catch (Exception e) {
   System.out.println("Error: " + e);
}
```

Answers for Quiz on Chapter 12

Question 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 creates and starts a thread belonging to that class.

Answer:

The class that defines the thread:

```
public class CountingThread extends Thread {
   public static run() {
      for (int i = 1; i <= 10; i++)
         System.out.println(i);
   }
}
```

Code to create an object of type *CountingThread* and start the thread:

```
CountingThread counter;     // Declare a variable to
represent a thread.
counter = new CountingThread();  // Create the thread
object.
counter.start();                 // Start the thread
running.
```

(Note that it's possible to create a new thread and start it with one statement:

```
new CountingThread().start();
```

Because of the precedence of the `new` operator, this is equivalent to `(new CountingThread()).start()`, and the effect is to create a new object of type *CountingThread* and then to call the `start()` method in that object.)

Question 2:

Suppose that `thrd` is an object of type *Thread*. Explain the difference between calling `thrd.start()` and calling `thrd.run()`.

Answer:

Calling `thrd.start()` creates a new thread of control and returns immediately after doing so; then the thread's `run()` method is called in the new thread of control. The code in the `run()` method is executed at the same time, in parallel, with the code that follows the call to `thrd.start()`.

The statement `thrd.run()` calls the run method in the usual way: The `run()` method is executed in the same thread of control as the method that called `run()`. Only after the `run()` method returns will the code that follows `thrd.run()` be executed.

Question 3:

What is a *race condition*?

Answer:

A race condition is a problem that can occur in a multithreaded program. Suppose that a thread takes a sequence of actions in which one action can depend on the result of a previous action. A race condition occurs if it's possible for another thread to change or invalidate the result of the previous action, before the first thread completes the sequence. For example, there is a race condition in the simple assignment statement `count = count+1` because it is executed as a sequence of steps. The old value of `count` is read, one is added to that value, and the new value is stored back into `count`. A race condition occurs if it is possible for another thread to increment the value stored in `count` between the time when the first thread reads the old value and the time when it stores the new value. In this case, the race

condition can result in an incorrect value for `count`—`count` might be increased by one when it was supposed to be increased by two. Another example occurs in the if statement

```
if ( ! list.isEmpty() )
    return list.removeFirst();
```

There is a race condition if it is possible for another thread to empty the list between the time when the first thread tests whether the list is empty and the time when the first thread tries to remove an element from the list. In this case, the race condition can result in an exception when the first thread tries to remove an item from an empty list.

Question 4:

How does synchronization prevent race conditions, and what does it mean to say that synchronization only provides *mutual* exclusion?

Answer:

Synchronization makes it possible for a thread to complete a sequence of actions without interference from another thread. Two threads cannot be synchronized on the same object at the same time. While one thread is executing a synchronized block of code, it's impossible for another thread to be executing the same block of code, or any other block of code that is synchronized on the same object. For example, if a thread executes

```
synchronized(list) {
   if ( ! list.isEmpty() )
      return list.removeFirst();
}
```

then, assuming that all code that manipulates `list` is properly synchronized, it can be sure that no other thread will be able to empty the list between the time when the first thread tests whether the list is empty and the time when it removes the first element from the list.

Synchronization provides only *mutual* exclusion because it only protects a thread from other threads that are *also synchronized* on the same object. In the example, the code has exclusive access to `list` only if all the code segments that manipulate `list` are synchronized. There is no protection against a thread that executes the statement `list.clear()` without synchronization.

Question 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?

Answer:

The execution time will depend on whether the program is being run on a computer that has more than one processor. If so, the execution time could be as little as 2 seconds, since each of two processors can do half of the 4-seconds worth of work. If the computer has only one processor, however, the two-threaded program will still take 4 seconds, since all the work will have to be done by the single processor. (Even on a multiprocessor computer, it might take longer than 2 seconds, if the work is not divided evenly between the two threads.)

Question 6:

What is an *ArrayBlockingQueue* and how does it solve the producer/consumer problem?

Answer:

An *ArrayBlockingQueue* is a queue in which the operations of adding and removing items can block. Adding an item will block if the queue is full; removing an item will block if the queue is empty. (Furthermore, operations on the queue are properly synchronized for use in a multithreaded program.)

The producer/consumer problem is the problem of safely and efficiently getting items that are produced by one group of threads to a second group of threads that consume the items. If the items are sent through a blocking queue, then the threads in the second group will block when there are no items available for them to consume, and threads in the first group will block if they are producing items faster than they can be consumed. (Furthermore, the synchronization guarantees that no item will be lost or consumed twice.)

In an application in which only the consuming threads should block, a *LinkedBlockingQueue*, which has unlimited capacity, can be used.

Question 7:

What is a *thread pool*?

Answer:

Thread pools are used when a large number of tasks are to be performed, as an alternative to creating a new thread to execute each task. A thread pool is a relatively small collection of threads that are available for performing tasks. When a task becomes available, instead of creating a new thread for the task, the task is assigned to one of the threads from the pool. When the task is complete, the thread goes back into the pool so that more tasks can be assigned to it.

Typically, tasks are placed into a queue as they become available. Each thread in the pool runs in an infinite loop in which it repeatedly takes a task from the queue and executes it. (Blocking queues work well for this application.)

Question 8:

Network server programs are often *multithreaded.* Explain what this means and why it is true.

Answer:

A multi-threaded server uses threads to handle the client connections that it accepts. A server program is generally designed to process connection requests from many clients. It runs in an infinite loop in which it accepts a connection request and processes it. If the processing takes a significant amount of time, it's not a good idea to make the other clients wait while the current client is processed. The solution is for the server to make a new thread to handle each client connection, or to use a thread pool of threads that can handle the client connections. The server can continue to accept more client connections even while the first client is being serviced.

Question 9:

Why does a multithreaded network server program often use many times more threads than the number of available processors?

Answer:

Network operations can *block*, which means that a thread that handles communication with a client will often spend most of its time sleeping. In order to keep all the processors busy, the number of **active** threads should be comparable to the number of processors. If at any given time, the number of active threads is only a fraction of the total number of threads, then the total number of threads should be several times the number of processors.

Question 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.)

Answer:

The `getValue()` method has to be synchronized because of the caching of local data that was discussed in Subsection 12.1.4. If `getValue()` were not synchronized, it is possible that a thread that calls `getValue()` would see an old, cached value of `count` rather than the most current value. Synchronization ensures that the most current value of `count` will be seen. If `count` were declared to be a `volatile` variable, then `getValue()` would not have to be synchronized. However, `increment()` would still need to be synchronized to prevent the race condition.

Answers for Quiz on Chapter 13

Question 1:

What is an "observable property"?

Answer:

An observable **value** is an object that emits events when its value is changed. It is possible to add listeners to the object, and the listeners will be notified when its value changes.

An observable property is a property (that is, an instance variable) in an object that is an observable value. Furthermore, the use of the term "observable property" usually implies that it is possible to "bind" other properties of the same type to this one, so that the value of the bound property will track the value of the property to which it is bound.

Question 2:

Suppose that `input` is a *TextField* and that `label` is a *Label*. Suppose that you want the text on the label to always be the same as the text in the text field. Write **two** code segments to accomplish that, one using an event listener and one using property binding.

Answer:

The text field has a property `input.textProperty()` that represents the text in the input box. We can add an event listener to that property, and the handler for that event can set the text in the label:

```
input.textProperty().addListener( e ->
label.setText(input.getText()) );
```

The label also has a property, `label.textProperty()`, representing the text that it displays, so another option is to bind that property to the property of the input box:

```
label.textProperty().bind( input.textProperty() );
```

One difference between these two approaches is that the binding will immediately copy the current text from the text field into the label. When using the listener, we would have to manually set up the initial texts to be the same.

By the way, the listener that was used in the above command is an *InvalidationListener*. Another approach would be to use a *ChangeListener*. The handler for a *ChangeEvent* takes three parameters, where the third parameter is the new value of the observable property, so the task could also be accomplished by

```
input.textProperty().addListener( (o,oldVal,newVal) ->
label.setText(newVal)) );
```

Question 3:

Describe the picture that is produced by the following code, where `canvas` is a *Canvas*:

```
GraphicsContext g = canvas.getGraphicsContext2D();
g.setFill(Color.WHITE);
g.fillRect(0,0,canvas.getWidth(),canvas.getHeight());
g.translate( canvas.getWidth()/2, canvas.getHeight()/2 );
g.rotate( 30 );
g.setFill(Color.RED);
g.fillRect(0,0,100,100);
```

Answer:

The entire canvas is filled with white, except for a filled red square that is 100-by-100 pixels in size. The corner of the square is at the center of the canvas that is being painted, and the top side of the square descends at a 30

degree angle from that point. The command `g.fillRect(0,0,100,100)` draws a red square with its top left corner at (0,0). But that square is first rotated by 30 degrees and then translated. The translation moves the point (0,0) to the point `(canvas.getWidth()/2,canvas.getHeight()/2)`, placing the top left corner of the square at the center of the canvas. (Transformations are applied to objects in the opposite of their order in the code.)

Of course, this answer assumes that no transformation had been applied **before** this code is executed. If there is a previous transformation, then that transform would be applied to the picture that is described in the answer, before that picture appears on the screen.

Question 4:

Create a *LinearGradient* paint and use it to fill the rectangle drawn by `g.fillRect(100,100,300,200)`. The rectangle should look like this, light gray at the top and black at the bottom:

(The API for creating gradient paints is complicated. It's OK to look it up!)

Answer:

The gradient can be defined by two color stops, one at 0, with color `Color.LIGHTGRAY` and one at 1, with color `Color.BLACK`. Since the

rectangle is covered by one copy of the gradient, the `CycleMethod` for the gradient is irrelevant. We can specify the start point for the gradient to be the top left corner of the rectangle, (100,100), and the end point for the gradient to be the bottom left corner, (100,300). (Note that the height of the rectangle is 200; since the top of the rectangle is at y=100, the bottom is at y=100+200. The end point of the gradient is specified using (x,y)-coordinates, not width and height like the rectangle.) So the rectangle can be drawn with

```
LinearGradient gradient =
                    new LinearGradient( 100,100, 100,300,
false, CycleMethod.REPEAT,
                          new Stop(0,Color.LIGHTGRAY), new
Stop(1,Color.BLACK));
g.setFill(gradient);
g.fillRect(100,100,300,200)
```

Many variations on this are possible. For example, the start point could be at the bottom right corner, and the end point at the top right corner. In that case, the color for the first color stop would be black, and the color for the second color stop would be light gray. We could also use a proportional gradient, where the coordinates for the start and end points are given in a coordinate system where the coordinates of the top left corner are (0,0) and the coordinates of the bottom left corner are (0,1):

```
LinearGradient gradient =
                    new LinearGradient( 0,0, 0,1, true,
CycleMethod.REPEAT,
                          new Stop(0,Color.LIGHTGRAY), new
Stop(1,Color.BLACK));
```

Question 5:

Suppose that `g` is a *GraphicsContext*. Explain the purpose of the methods `g.save()` and `g.restore()`.

Answer:

A call to `g.save()` pushes a copy of the current state of the graphics context onto a stack of saved states. The state include properties of the graphics context that affect drawing such as the transform, the stroke and fill paints, and the line width. A call to `g.restore()` will pop the most recently saved state from the stack and restore graphics context properties to their values from that state. Any changes to the state that were made after the corresponding call to `g.save()` will be undone. This allows you to make temporary changes to the state while being sure that those changes will not carry over to future drawing.

Question 6:

What does the acronym *MVC* stand for, and how does it apply to the *List* class?

Answer:

MVC stands for "Model-View-Controller." In a *List*, the *view* is the actual visible component on the screen. The *model* is the collection of data that specifies, among other things, what appears in each cell of the list. This data is stored in an observable list, and the view listens for changes in that list and redraws itself when any change occurs. (Another part of the model is the "selection model", which keeps track of which item is currently selected in the list.) The *controller* is responsible for interaction with the user. It consists mostly of a bunch of listener objects that listen for events generated when the user interacts with the list. The listeners respond by making changes in the model, which will in turn cause a change in the view.

Question 7:

What is the difference between a "modal" dialog box and a "modeless" dialog box?

Answer:

When a modal dialog box is shown, the user will not be able to use the parent window of the dialog box until the dialog box is closed. With a modeless dialog box, it is possible for the user to work with both the dialog box and its parent window. (Some dialog boxes are "application modal," meaning that they block interaction with the entire application, not just with one window.)

Question 8:

The Java API includes some classes in a package named `org.w3c.dom`. Why such a funny package name?

Answer:

The packages presumably come from an organization that owns the Web domain name `w3c.org`. (In fact, it's the organization that controls standards for the web.) This follows the recommended naming convention for packages: The package name starts with the pieces of the domain name in reverse order. The last part of the package name, "dom", doesn't come from the domain name; it was something chosen by the organization for this particular package, to distinguish it from other packages that might be put out by the same organization. (In fact, the `org.w3c.dom` package contains classes and subpackages related to the "Document Object Model," which is used in the processing of XML and HTML documents.)

Question 9:

Suppose that `closeItem` is a *MenuItem*. What is done by the following statement? (What is an "accelerator"?)

```
closeItem.setAccelerator( KeyCombination.valueOf("ctrl+W")
);
```

Answer:

An accelerator for a menu item is a combination of keyboard keys that is equivalent to selecting that item from a menu. `KeyCombination.valueOf("ctrl+W")` is a key combination that represents pressing the "W" key while holding down the control key. Since that key combination is installed as the accelerator for `closeItem`, hitting "W" while holding down the control key will invoke the same action as selecting `closeItem` with the mouse.

Question 10:

What is meant by *Internationalization* of a program?

Answer:

Internationalization refers to writing the program in a way that will make it easy to adapt the program for use in a variety of "locales." For example, it should be easy to translate all the strings that are used in the program into other languages. To make this possible the strings should not be hard coded into the program itself. Instead, they are placed in a separate resource file, so that the program can be translated into another language simply by writing a resource file for that language. (Internationalization also applies to the format that is used for dates and numbers.)

Answers to End-Of-Chapter Exercises

This section contains sample solutions to the exercises that occur at the end of each chapter, starting with Chapter 2. The solution includes a discussion of the problem, as well as the code for a solution. The discussion is meant to be read, and in many cases will supplement the reading from the chapter itself.

Remember that there are many ways to solve a programming problem!

Solution for Programming Exercise 2.1

Exercise 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:

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

Discussion

This is a very simple program. It consists of nothing but a sequence of `System.out.println` statements. The hard part is designing the letters. You have to experiment in your text editor until you get letters that look right to you. Add `System.out.println("` at the beginning of the first line, then copy-and-paste it to the beginning of each of the other lines. (You don't need to type it out on each line! Creative use of copy-and-paste can save you a lot of work.) Then add `");` to the end of each line. In my program, I've also added a `System.out.println()` statement at the beginning and the end. The extra blank lines make the output look better.

Don't forget that in order to have a complete program, you have to put the program statements in a `main` subroutine and put that subroutine in a `public class`. Also, the name of the file must correspond to the name of the class; in this case, the file must be named PrintInitials.java.

The Solution

```
    public class PrintInitials {

       /*  This program prints my initials (DJE) in big
letters,
           where each letter is nine lines tall.
       */

       public static void main(String[] args) {
          System.out.println();
          System.out.println("           ******
*************        **********");
          System.out.println("           **    **
**            **");
          System.out.println("           **     **
**            **");
          System.out.println("           **      **
**            **");
          System.out.println("           **      **
**            ********");
          System.out.println("           **      **
**     **             **");
          System.out.println("           **     **
**    **             **");
          System.out.println("           **    **
**   **              **");
          System.out.println("           *****
****               **********");
          System.out.println();
       }  // end main()

    }  // end class
```

Solution for Programming Exercise 2.2

Exercise 2.2:

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 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
```

Discussion

When designing a program, one of the first things you should ask yourself is, "What values do I need to represent?" The answer helps you decide what variables to declare in the program. This program will need some variables to represent the numbers showing on each die and the total of the two dice. Since these numbers are all integers, we can use three variables of type int. I'll call the variables `die1`, `die2`, and `roll`. The program begins by declaring the variables:

```
int die1;
int die2;
int roll;
```

In the actual program, of course, I've added a comment to explain the purpose of each variable. The values of `die1` and `die2` can be computed using the expression given in the exercise:

```
die1 = (int)(Math.random()*6) + 1;
die2 = (int)(Math.random()*6) + 1;
```

Note that even though the expressions on the right-hand sides of these assignment statements are the same, the values can be different because the function `Math.random()` can return different values when it is called twice.

We can then compute `roll = die1 + die2` and use three `System.out.println` statements to display the three lines of output:

```
System.out.println("The first die comes up " + die1);
System.out.println("The second die comes up " + die2);
System.out.println("Your total roll is " + roll);
```

Note that I've chosen to use the concatenation operator, `+`, to append the value of `die1` onto the string "The first die comes up". Alternatively, I could use two output statements:

```
System.out.print("The first die comes up ");
System.out.println(die1);
```

Yet another possibility is to use `System.out.printf`:

```
System.out.printf("The first die comes up %d%n", die1);
```

I'll also note that I could get away without the variable `roll`, since I could output the value of the expression `die1 + die2` directly:

```
System.out.println("Your total roll is " + (die1 + die2));
```

However, it's generally better style to have a meaningful name for a quantity. By the way, the parentheses around `(die1 + die2)` are essential because of the precedence rules for the `+` operator. You might try to experiment with leaving them out and see what happens.

As a final variation, all three lines of output could be produced using a multiline text block. Text blocks were discussed in Subsection 2.3.4. Using a

text block also requires `System.out.printf`:

```
System.out.printf("""
    The first die comes up %d
    The second die comes up %d
    Your total roll is %d
    """, die1, die2, roll);
```

The Solution

```
public class RollTheDice {

    /*  This program simulates rolling a pair of dice.
        The number that comes up on each die is output,
        followed by the total of the two dice.
    */

    public static void main(String[] args) {

        int die1;   // The number on the first die.
        int die2;   // The number on the second die.
        int roll;   // The total roll (sum of the two
dice).

        die1 = (int)(Math.random()*6) + 1;
        die2 = (int)(Math.random()*6) + 1;
        roll = die1 + die2;

        System.out.println("The first die comes up " +
die1);
        System.out.println("The second die comes up " +
die2);
        System.out.println("Your total roll is " + roll);

    }  // end main()

}  // end class
```

Solution for Programming Exercise 2.3

Exercise 2.3:

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!".

Discussion

In order to read the name typed in by the user, this program uses one of the input routines from the non-standard *TextIO* class. So, this program can only be run if that class is available to the program, and the program should start with the import directive `import textio.TextIO`. I will use `System.out` for output. (Although it is possible to use `TextIO` for output as well as for input, I will generally avoid that option in my examples.)

A name is a sequence of characters, so it is a value of type *String*. The program must declare a variable of type *String* to hold the user's name. I declare another variable to hold the user's name with all the letters in the name converted to upper case. The conversion might be difficult, except that *String* objects have a function that lets you do the conversion with one line of code. If `usersName` is the variable that refers to the name that the user enters, then the function call `usersName.toUpperCase()` returns the string obtained by replacing any lower case letters in the name with upper case letters.

There are several functions in the *TextIO* class that can be used for reading `Strings`: `getWord()`, `getlnWord()`, and `getln()`. The first two routines only read a single word, so if the user entered "David J. Eck", they would only read the first name, "David". The `getln()` routine will read the entire line, and so would get the whole name. For this program, I use `getln()`, but you might prefer to use just the first name.

I also give a version of the program that uses the built-in class *Scanner*, instead of *TextIO*, for reading input from the user. See Subsection 2.4.6 for a discussion of the *Scanner* class. In the second version of the program, a *Scanner* named `stdin` is created, and the user's input is read using the function `stdin.nextLine()`, which has essentially the same functionality as `TextIO.getln()`.

(For this program, by the way, `TextIO.getWord()` and `TextIO.getlnWord()` would be equivalent. They return the same value. The second version of the routine, `getlnWord()`, would then discard the rest of the user's line of input. However, since this program is only doing one input operation, it doesn't matter whether it's discarded. It would only matter when it came time to read a second value from input.)

The Solution

```
import textio.TextIO;

public class Greeting {

    /*  This program asks the user's name and then
        greets the user by name.  This program depends
        on the non-standard class, TextIO.
    */

    public static void main(String[] args) {

        String usersName;       // The user's name, as
entered by the user.
        String upperCaseName;   // The user's name,
converted to upper case letters.

        System.out.print("Please enter your name: ");
        usersName = TextIO.getln();

        upperCaseName = usersName.toUpperCase();

        System.out.println("Hello, " + upperCaseName + ",
nice to meet you!");

    }  // end main()
```

```
}  // end class
```

A version using *Scanner* for input:

```
import java.util.Scanner;

public class GreetingWithScanner {

    public static void main(String[] args) {

        Scanner stdin = new Scanner( System.in );

        String usersName;      // The user's name, as
entered by the user.
        String upperCaseName;  // The user's name,
converted to upper case letters.

        System.out.print("Please enter your name: ");
        usersName = stdin.nextLine();

        upperCaseName = usersName.toUpperCase();

        System.out.println("Hello, " + upperCaseName + ",
nice to meet you!");

    }  // end main()

}  // end class
```

Solution for Programming Exercise 2.4

Exercise 2.4:

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.

Discussion

The program will need variables to represent the number of each type of coin. Since the number of coins has to be an integer, these variables are of type int. I'll call the variables `quarters`, `dimes`, `nickels`, and `pennies`.

The total value of the coins, when expressed in dollars, can be a non-integer number such as 1.57 or 3.02. Since the total value in dollars is a real number, I will use a variable of type double to represent it. The variable is named `dollars`

The outline of the program is clear enough:

```
Declare the variables.
Ask the user for the number of each type of coin, and read
the responses.
Compute the total value of the coins, in dollars.
Display the result to the user.
```

The function `TextIO.getlnInt()` can be used to read each of the user's responses. The alternative function `TextIO.getInt()` could also be used, but it is less safe. Suppose, for example, that the user responds to the request to type in the number of quarters by entering "7 quarters". After `TextIO.getlnInt()` reads the number 7, it will discard the extra input "quarters". `TextIO.getInt()` will read the 7 correctly, but the extra input is not discarded. Later, when the program tries to read the number of dimes, it

sees the left-over input and tries to read that, without giving the user a chance to type in another response. You might want to experiment and see what happens if you change `getlnInt()` to `getInt()`. (Of course, if the user's response is "I have 7 quarters" or "seven", then you are out of luck in any case.)

Since one quarter is worth 0.25 dollars, the number of dollars in N quarters is 0.25*N. Similarly, a dime is worth 0.10 dollars, a nickel is 0.05 dollars, and a penny is 0.01 dollars. So, to get the total value of all the user's coins, I just have to add up `(0.25*quarters) + (0.10*dimes) + (0.05*nickels) + (0.01*pennies)`. This value is assigned to the variable, `dollars`, and that is the result that is displayed to the user. Formatted output, using `System.out.printf`, can be used to specify that the output value should be printed with two digits after the decimal point.

Alternatively, I could first have computed the total number of cents in all the coins, and then divided by 100 to convert the amount into dollars:

```
int totalCents;   // Total number of cents in the coins.

totalCents = 25*quarters + 10*dimes + 5*nickels + pennies;
dollars = totalCents/100.0;
```

Since `totalCents` is of type int, it is essential here that I compute `dollars` as `totalCents/100.0` and not as `totalCents/100`. The value computed by `totalCents/100` is an integer. For example, if `totalCents` is 397, then `totalCents/100` is 3. Using `totalCents/100.0` forces the computer to compute the answer as a real number, giving 3.97.

A second version of the program, using a *Scanner* for input, is also given. (See Subsection 2.4.6.) A *Scanner* named `stdio` is created, and integers are read using the function `stdio.nextInt()`. This method corresponds to `TextIO.getInt()`, not `TextIO.getlnInt()`. To get something equivalent to `TextIO.getlnInt()`, we can follow the call to `stdio.nextInt()` with a call to `stdio.nextLine()`. The call to `stdio.nextLine()` will read and discard any extra characters on the same line after the integer that was read by `stdio.nextInt()`. If you are not worried about the user typing extra stuff on a line, you can leave out the calls to `stdio.nextInt()`.

The Solution

```
import textio.TextIO;

public class CountChange {

   /*  This program will add up the value of a number of
quarters,
       dimes, nickels, and pennies.  The number of each
type of
       coin is input by the user.  The total value is
reported
       in dollars.  This program depends on the non-
standard class,
       TextIO.
   */

   public static void main(String[] args) {

      int quarters;   // Number of quarters, to be input by
the user.
      int dimes;      // Number of dimes, to be input by
the user.
      int nickels;    // Number of nickels, to be input by
the user.
      int pennies;    // Number of pennies, to be input by
the user.

      double dollars; // Total value of all the coins, in
dollars.

      /* Ask the user for the number of each type of coin.
*/

      System.out.print("Enter the number of quarters:  ");
      quarters = TextIO.getlnInt();

      System.out.print("Enter the number of dimes:     ");
      dimes = TextIO.getlnInt();

      System.out.print("Enter the number of nickels:   ");
      nickels = TextIO.getlnInt();

      System.out.print("Enter the number of pennies:   ");
      pennies = TextIO.getlnInt();
```

```
      /* Add up the values of the coins, in dollars. */

      dollars = (0.25 * quarters) + (0.10 * dimes)
                          + (0.05 * nickels) + (0.01 *
pennies);

      /* Report the result back to the user. */

      System.out.println();
      System.out.print("The total in dollars is $");
      System.out.printf("%1.2f", dollars);  // Formatted
output!
      System.out.println();

   }  // end main()

}  // end class
```

A version using *Scanner* for input:

```
import java.util.Scanner;

public class CountChangeWithScanner {

   public static void main(String[] args) {

      int quarters;   // Number of quarters, to be input by
the user.
      int dimes;      // Number of dimes, to be input by
the user.
      int nickels;    // Number of nickels, to be input by
the user.
      int pennies;    // Number of pennies, to be input by
the user.

      Scanner stdio = new Scanner( System.in );

      double dollars; // Total value of all the coins, in
dollars.

      /* Ask the user for the number of each type of coin.
*/

      System.out.print("Enter the number of quarters:  ");
      quarters = stdio.nextInt();
      stdio.nextLine();
```

```
      System.out.print("Enter the number of dimes:     ");
      dimes = stdio.nextInt();
      stdio.nextLine();

      System.out.print("Enter the number of nickels:   ");
      nickels = stdio.nextInt();
      stdio.nextLine();

      System.out.print("Enter the number of pennies:   ");
      pennies = stdio.nextInt();
      stdio.nextLine();

      /* Add up the values of the coins, in dollars. */

      dollars = (0.25 * quarters) + (0.10 * dimes)
                          + (0.05 * nickels) + (0.01 *
pennies);

      /* Report the result back to the user. */

      System.out.println();
      System.out.print("The total in dollars is $");
      System.out.printf("%1.2f", dollars);  // Formatted
output!
      System.out.println();

   }  // end main()

}  // end class
```

Solution for Programming Exercise 2.5

Exercise 2.5:

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.

Discussion

All the variables for the program will be of type int. The quantities that we have to represent are: the number of eggs, the number of dozens in that many eggs, and the number of left over eggs. I will declare variables named `eggs`, `dozens`, and `extras` to represent these quantities.

The number of eggs can be read from the user's input using `eggs = TextIO.getlnInt()`.

The description of the problem already tells us that `dozens` can be computed as `eggs/12` and `extras` can be computed as `eggs%12`. So, the first version of the program is easy to write. Note that the entire output from this program could have been done with one `System.out.printf`:

```
System.out.print("Your number of eggs is %d dozen and %d
extras%n",
```

```
                                        dozens, extras);
```

The improved version requires a little thought. Given a pile of eggs that you want to divide into gross, dozens, and extras, you could first remove as many gross as possible, leaving a pile of between 0 and 143 eggs. This is just like dividing the pile into dozens, except that the unit of measure is 144 eggs instead of 12. So, `eggs/144` will give the number of gross, and `eggs%144` is the number of eggs left over. Next, the eggs in the left-over pile still have to be divided into dozens, plus some extra number of eggs between 0 and 11. Of course, we already know how to do this: If there are `N` eggs in the pile, the number of dozens is `N/12` and the number of extra eggs is `N%12`. This leads to the computation in the improved version of the program, shown below. (Note that this program uses the variable named `aboveGross` to represent a quantity that is not mentioned explicitly in the problem description. This is far from being unusual.)

The Solution

```
import textio.TextIO;

public class Dozens {

   /*  This program will convert a given number of eggs
into
       the number of dozens plus the number of left over
eggs.
       For example, 57 eggs is 4 dozen eggs plus 9 eggs
left over.
       The number of eggs is input by the user, and the
computed
       results are displayed.
   */

   public static void main(String[] args) {

      int eggs;    // Number of eggs, input by user.
      int dozens;  // How many dozens in that number of
eggs?
      int extras;  // How many eggs are left over, above an
integral
                   //    number of dozens?  This value is
in the
```

```
                    //    range 0 to 11, and it is computed
as
                    //    the remainder when eggs is divided
by 12.

      System.out.print("How many eggs do you have?  ");
      eggs = TextIO.getlnInt();

      dozens = eggs / 12;
      extras = eggs % 12;

      System.out.print("Your number of eggs is ");
      System.out.print(dozens);
      System.out.print(" dozen and ");
      System.out.print(extras);
      System.out.println();

   }  // end main()

}  // end class
```

Improved version:

```
import textio.TextIO;

public class GrossAndDozens {

   /*  This program will convert a given number of eggs
into
       the number of gross plus the number of dozens plus
the
       number of left over eggs.
          For example, 1342 eggs is 9 gross plus 3 dozen
plus 10.
       The number of eggs is input by the user, and the
computed
       results are displayed.
   */

   public static void main(String[] args) {

      int eggs;         // Number of eggs, input by user.
      int gross;        // How many gross in that number of
eggs?
      int aboveGross;   // How many eggs are left over,
above an
                        //    integral number of gross?
```

```
This number
                        //    can be computed as eggs %
144, and is
                        //    in the range 0 to 143.  This
number will
                        //    be divided into dozens and
extras.
      int dozens;       // How many dozens in aboveGross?
      int extras;       // How many eggs are left over,
above integral
                        //    numbers of gross and dozens?

      System.out.print("How many eggs do you have?  ");
      eggs = TextIO.getlnInt();

      gross = eggs / 144;
      aboveGross = eggs % 144;

      dozens = aboveGross / 12;
      extras = aboveGross % 12;

      System.out.print("Your number of eggs is ");
      System.out.print(gross);
      System.out.print(" gross, ");
      System.out.print(dozens);
      System.out.print(" dozen, and ");
      System.out.print(extras);
      System.out.println();

   }  // end main()

}  // end class
```

Solution for Programming Exercise 2.6

Exercise 2.6:

This exercise asks you to write a program that tests some of the built-in subroutines for working with *Strings*. The program should ask the user to enter their first name and their last name, separated by a space. Read the user's response using `TextIO.getln()`. Break the input string up into two strings, one containing the first name and one containing the last name. You can do that by using the `indexOf()` subroutine to find the position of the space, and then using `substring()` to extract each of the two names. Also output the number of characters in each name, and output the user's initials. (The initials are the first letter of the first name together with the first letter of the last name.) A sample run of the program should look something like this:

```
Please enter your first name and last name, separated by a
space.
? Mary Smith
Your first name is Mary, which has 4 characters
Your last name is Smith, which has 5 characters
Your initials are MS
```

Discussion

Once you have the first name and the last name in separate *String* variables, it's easy to produce the desired output. The length of a string, `str`, can be obtained by calling the function `str.length()`. To get the first character of `str`, you can call `str.charAt(0)`. (Remember that characters are numbered starting from zero.) If the first and last names are in the variables `firstName` and `lastName`, then the output of the program can be produced using

```
System.out.println("Your first name is " + firstName + ",
which has "
                           + firstName.length() + "
characters.");
System.out.println("Your last name is " + lastName + ",
```

```
which has "
                            + lastName.length() + "
characters.");
System.out.println("Your initials are " +
firstName.charAt(0) + lastName.charAt(0));
```

Or, using formatted output to make it easier to read:

```
System.out.printf( "Your first name is %s, which has %d
characters.%n",
                                    firstName,
firstName.length() );
System.out.printf( "Your last name is %s, which has %d
characters.%n",
                                    lastName,
lastName.length() );
System.out.printf( "Your initials are %s%s%n",
                                    firstName.charAt(0),
lastName.charAt(0) );
```

Note that it is not necessary to put values like `firstName.length()` and `firstName.charAt(0)` into their own variables, if all that you want to do with them is print them out. (But some people might consider it better style to use variables.)

The harder part of the problem is breaking up the input string to get the first and last name. If the user's input is in a *String* named `input`, then `input.indexOf(' ')` is an int that gives the position of the space in the string. (Note that the space character is written as a literal that consists of a space between two left single quotation marks.)

Once you have the position of the space, you have to figure out how to use it to get the first and last name. Let's look at exactly how the characters in a string are numbered:

```
M  a  r  y     S  m  i  t  h
0  1  2  3  4  5  6  7  8  9
```

In this example, the space is at index 4. We have to extract the substring in positions 0 through 3 and the substring in positions 5 through 9. The value of `input.substring(start,end)` will be a string that starts in position `start` and ends in position `end-1`. Note that the value of the second parameter is the

position **after** the end of the substring. So, the fist name in this case is `input.substring(0,4)` and the last name is `input.substring(5,10)`. (Even though there is no character in position 10, that's the correct value for the position after the end of the substring that we want.) In fact, there's an easier way to get the last name, using another version of `substring`: The value of `input.substring(start)` is the substring starting at position `start` and extending to the end of the string, so the last name in this example is `input.substring(4)`.

The important thing to note here is that the "4" in `input.substring(0,4)` is the index of the space character, and the "5" in `input.substring(5)` is that index plus 1. So we can get the first and last names using:

```
space = input.indexOf(' ');
firstName = input.substring(0,space);  // Everything
before the space
lastName = input.substring(space+1);   // Everything after
the space
```

By the way, this program will only work correctly if the user follows the instructions. If the user's input does not contain a space, then the value of `input.indexOf(' ')` will be -1, and the -1 will cause the program to crash when used in the substring function. And if the input doens't have exactly one space, then the program output won't be correct. For now, you don't have any way to test the user's input to see whether it is correct. To do that, you need to know about `if` statements, which you will learn about very soon.

One more remark: Instead of using `TextIO.getln()` to read a line of text and then breaking that line into two words, you could simply use `TextIO.getWord()` twice to read the two words from the same line of input —except that the exercise says to use `getln()`.

The Solution

```
import textio.TextIO;

/**
 * This program reads the user's first name and last name,
 * separated by a space.  It then prints the user's first
```

```
and
 * last names separately, along with the number of
characters
 * in each name.  It also prints the user's initials.  Note
that
 * this program will crash if the user's input does not
contain
 * a space.
 */
public class FirstNameLastName {

    public static void main(String[] args) {

        String input;     // The input line entered by the
user.
        int space;        // The location of the space in
the input.
        String firstName; // The first name, extracted from
the input.
        String lastName;  // The last name, extracted from
the input.

        System.out.println();
        System.out.println("Please enter your first name
and last name, separated by a space.");
        System.out.print("? ");
        input = TextIO.getln();

        space = input.indexOf(' ');
        firstName = input.substring(0, space);
        lastName = input.substring(space+1);

        System.out.println("Your first name is " +
firstName + ", which has "
                                  + firstName.length() + "
characters.");
        System.out.println("Your last name is " + lastName
+ ", which has "
                                  + lastName.length() + "
characters.");
        System.out.println("Your initials are " +
firstName.charAt(0) + lastName.charAt(0));

    }

}
```

Solution for Programming Exercise 2.7

Exercise 2.7:

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 that 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.

Discussion

TextIO can be used to read data from a file; this is discussed in Subsection 2.4.4. To read data from a file named "testdata.txt", all you need to do is say `TextIO.readFile("testdata.txt")`. From then on, the input functions in *TextIO* will read from the file instead of reading data typed in by the user. (Note that this assumes that the file is in the same directory with the program.) In this case, we can use `TextIO.getln()` to read the student's name from the first line of the file, and then we can read the exam grades by calling `TextIO.getln()` three times.

The average should be computed as a value of type double. Don't forget that if you divide an integer by an integer in Java, the result is an integer and the remainder of the division is discarded. To get the correct average in this case, the program divides the sum of the three grades by 3.0 rather than by 3.

One final technicality is that simply outputting a double value might print out something like 83.333333333333333. By default, all significant digits in the number are output. In this case, one digit after the decimal point is probably sufficient. The program uses formatted output to achieve this:

```
System.out.printf("The average grade for %s was %1.1f",
name, average);
```

The format string "The average grade for %s was %1.1f" is used to format the name and the average. The name is substituted for the format specifier `%s`, which means that the name is printed as a string, with no extra spaces. The average is substituted for `%1.1f`, which means that the average is printed as a floating point number with no extra spaces and with 1 digit after the decimal point.

You might want to run this program with no data file, or with a data file that is not in the correct format, to see what happens. (The program will crash and print an error message.)

The Solution

```
import textio.TextIO;

public class FindAverage {

   public static void main(String[] args) {

       String name;     // The student's name, from the
first line of the file.
       int exam1, exam2, exam3;     // The student's grades
on the three exams.
       double average;  // The average of the three exam
grades.

       TextIO.readFile("testdata.txt");  // Read from the
file.

       name = TextIO.getln();  // Reads the entire first
line of the file.
       exam1 = TextIO.getlnInt();
       exam2 = TextIO.getlnInt();
       exam3 = TextIO.getlnInt();

       average = ( exam1 + exam2 + exam3 ) / 3.0;

       System.out.printf("The average grade for %s was
%1.1f", name, average);
       System.out.println();
```

```
    }

}
```

Solution for Programming Exercise 3.1

Exercise 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.

Discussion

Since we want to roll the dice at least once, a `do..while` is appropriate. A pseudocode algorithm for the program is

```
Let countRolls = 0
do:
    roll the dice
    count this roll by adding 1 to countRolls
while the roll is not snake eyes
Output the value of countRolls
```

As in Exercise 2.2, we can simulate rolling one die by computing `(int)(Math.random()*6) + 1`.

We want to **stop** rolling the dice when the roll is a double 1. We want to **continue** rolling the dice while the roll is **not** a double 1. If `die1` and `die2` are variables representing the values of the dice, the condition for continuing to roll can be expressed as

```
while ( ! (die1 == 1 && die2 == 1) )
```

The exclamation point means "not", so the condition says that it is not the case that (both `die1` is 1 and `die2` is 1). That is, it is not the case that the

dice came up snake eyes. Another way to express the same condition is that at least one of the dice is **not** 1, that is, that either `die1` is not 1 **or** `die2` is not 1. In Java code, this is written:

```
while ( die1 != 1  ||  die2 != 1 )
```

This is the test that I use in my program. Students often get the `&&` and `||` operators mixed up, especially when negation is involved. (In this case, we could have avoided the problem by testing `while (die1+die2 != 2)`.)

Filling in some details gives an algorithm that can be easily converted into a program, which is shown below:

```
Let countRolls = 0
do:
    die1 = (int)(Math.random()*6) + 1
    die2 = (int)(Math.random()*6) + 1
    count this roll by adding 1 to countRolls
while die1 is not 1 or die2 is not 1
Output the value of countRolls
```

You could of course write the program using a regular `while` loop instead of `do..while`. But if you move the test to the start of the loop, you have to make sure that the variables are given values before you try to test those values. You need to "prime" the loop that exist only to ensure that it runs the first time. The pseudocode then becomes

```
Let countRolls = 0
Let die1 = 0 // Prime the loop with any value except 1
Let die2 = 0
while die1 is not 1 or die2 is not 1:
    die1 = (int)(Math.random()*6) + 1
    die2 = (int)(Math.random()*6) + 1
    count this roll by adding 1 to countRolls
Output the value of countRolls
```

Another option would be to use a "while(true)" loop. In that case, the test that is needed is the condition for stopping the loop, which is often easier to come up with than the test for continuing the loop, and no priming is necessary:

```
Let countRolls = 0
while true:
    die1 = (int)(Math.random()*6) + 1
    die2 = (int)(Math.random()*6) + 1
    count this roll by adding 1 to countRolls
    if die1 is 1 and die2 is 1
        break
Output the value of countRolls
```

The Solution

```
/**
 * This program simulates rolling a pair of dice until they
 * come up snake eyes.  It reports how many rolls were
needed.
 */

public class SnakeEyes {

   public static void main(String[] args) {

       int die1, die2;   // The values rolled on the two
dice.

       int countRolls;   // Used to count the number of
rolls.

       countRolls = 0;

       do {
          die1 = (int)(Math.random()*6) + 1;   // roll the
dice
          die2 = (int)(Math.random()*6) + 1;
          countRolls++;                        // and count
this roll
       } while ( die1 != 1 || die2 != 1 );

       System.out.println("It took " + countRolls + " rolls
to get snake eyes.");

   }  // end main()

}  // end class
```

Solution for Programming Exercise 3.2

Exercise 3.2:

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.

Discussion

Let's use a variable named `maxDivisors` to keep track of the largest number of divisors we have seen so far and use `numWithMax` to store the number that had that many divisors. We have to compute the number of divisors of each integer from 1 to 10000. Whenever we find a larger number of divisors than `maxDivisors`, we have to make note of that fact by changing the values of `maxDivisors` and `numWithMax`. At the end of the process, `maxDivisors` will be the absolute maximum number of divisors and `numWithMax` will be a number that had that many divisors. These are the values we want to print out. We can express this with a pseudocode algorithm

```
for each integer N from 1 to 10000:
   Count the number of divisors of N
   If that number is greater than maxDivisors:
       Let maxDivisors = the number of divisors of N
       Let numWithMax = N
Output maxDivisors and numWithMax
```

However, there is a problem here: The very first time `maxDivisors` is used in the test "If that number is greater than maxDivisors," the variable `maxDivisors` hasn't yet been assigned a value. The computer will report this as an error. This can be fixed by assigning a value to `maxDivisors` before the beginning of the `for` loop. One way to do this is to handle `N=1` as a special case before the loop and then to let `N` go from 2 to 10000 in the for loop. We know that `N=1` has just 1 divisor:

```
Let maxDivisors = 1  // number of divisors of 1
Let numWithMax = 1
for each integer N from 2 to 10000:
   Count the number of divisors of N
   If that number is greater than maxDivisors:
       Let maxDivisors = the number of divisors of N
       Let numWithMax = N
Output maxDivisors and numWithMax
```

Here's a curious thing: If you leave out the line "`numWithMax = 1`" from the program, the computer will report a syntax error where you try to output the value of `numWithMax`. It will say that the variable `numWithMax` might not have been initialized. That is, it might never have been assigned a value. Now, **you** know that it will be assigned a value (since when `N=2` is processed, `numWithMax` will become 2). However, when the computer compiles the program, **it** doesn't know whether the body of the `if` statement will ever be executed, so it doesn't know whether `numWithMax` will ever be assigned a value. The syntax rule is that every variable must be "definitely assigned" before its value is used. This means that it is assigned a value on every possible execution path through the program. Definite assignment was discussed in Subsection 3.1.5.

We still have to expand the step "Count the number of divisors of N." This was already done in Subsection 3.4.2 for the example program `CountDivisors.java`. This step requires another `for` loop, so we have here an example of one `for` loop nested inside another. Here is a complete algorithm, which can be translated into a program:

```
Let maxDivisors = 1  // number of divisors of 1
Let numWithMax = 1
for each integer N from 2 to 10000:
    Let divisorCount = 0
```

```
    for each D from 1 to N:
        if D is a divisor of N:
            add 1 to divisorCount
    If divisorCount is greater than maxDivisors:
        Let maxDivisors = the number of divisors of N
        Let numWithMax = N
Output maxDivisors and numWithMax
```

This can be translated pretty much directly into a program. By the way, the maximum number of divisors is 64. There are two numbers between 1 and 10000 that have 64 divisors, 7560 and 9240. The program will output the first of these. (It would output the second if the test "`if (divisorCount > maxDivisors)`" were changed to "`if (divisorCount >= maxDivisors)`". Do you see why?)

The Solution

```
/**
 * This program finds an integer between 1 and 10000 that
has
 * the largest number of divisors.  It prints out the
maximum
 * number of divisors and an integer that has that many
divisors.
 */

public class MostDivisors {

   public static void main(String[] args) {

       int N;             // One of the integers whose
divisors we have to count.
       int maxDivisors;   // Maximum number of divisors seen
so far.
       int numWithMax;    // A value of N that had the given
number of divisors.

       maxDivisors = 1;   // Start with the fact that 1 has
1 divisor.
       numWithMax = 1;

       /* Process all the remaining values of N from 2 to
10000, and
          update the values of maxDivisors and numWithMax
```

```
whenever we
          find a value of N that has more divisors than the
current value
          of maxDivisors.
       */

       for ( N = 2;  N <= 10000;  N++ ) {

           int D;  // A number to be tested to see if it's
a divisor of N.
           int divisorCount;  // Number of divisors of N.

           divisorCount = 0;

           for ( D = 1;  D <= N;  D++ ) {  // Count the
divisors of N.
              if ( N % D == 0 )
                 divisorCount++;
           }

           if (divisorCount > maxDivisors) {
              maxDivisors = divisorCount;
              numWithMax = N;
           }

       }

       System.out.println("Among integers between 1 and
10000,");
       System.out.println("The maximum number of divisors
is " + maxDivisors);
       System.out.println("A number with " + maxDivisors +
" divisors is " + numWithMax);

   } // end main()

}
```

Solution for Programming Exercise 3.3

Exercise 3.3:

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 consists 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.

Discussion

We need a loop to read and evaluate expressions. It's easiest to use a `break` statement to end the loop at the appropriate time:

```
Repeat indefinitely:
    Get the user's input.
    if the first number is 0:
        Break out of the loop
    Find the value of the expression
    Display the value.
```

Getting the user's input involves reading three data values. We need three variables to store these values. It's best to test whether the first number is 0 right after we read it, so the user will just have to type a 0 to end the program, not a complete expression such as 0 + 0. "Repeat indefinitely" can be written as "`while (true)`":

```
while (true):
    Let firstNum = TextIO.getDouble()
    if firstNum is 0:
        Break out of the loop
    Let operator = TextIO.getChar()
```

```
    Let secondNum = TextIO.getlnDouble()
    Find the value of the expression
    Display the value.
```

To evaluate the user's expression, we have to test the `operator` to find out which operation to compute. We can do this with either a multiway `if` statement or with a `switch` statement. In the program below, I use a `switch`. The `if` statement would be:

```
if ( operator == '+' )
   value = firstNum + secondNum;
else if ( operator == '-' )
   value = firstNum - secondNum;
else if ( operator == '*' )
   value = firstNum * secondNum;
else if ( operator == '/' )
   value = firstNum / secondNum;
else {
   System.out.println("Unknown operator: " + operator);
   continue;  // back to start of loop
}
```

The computer won't let you get away without the `else` part of the `if` statement or the `default` case in the `switch`, since that would leave a possibility that the variable, `value`, is not assigned a value before it is printed out, which violates the definite assignment rule. Note that char values are tested for equality using the `==` operator. Chars are not Strings! Strings would have to be compared using the `equals()` method from the *String* class; chars however are primitive type values, and they don't even include any methods.

This program could be improved by having it print out an error message if the user tries to divide by zero.

(This program could use a *Scanner* instead of *TextIO* for input, replacing `TextIO.getDouble()` with `stdin.nextDouble()` and `TextIO.getChar()` with `stdin.next()`, where `stdin` is the *Scanner*. (See Subsection 2.4.6) One difference between the behavior of the two versions of the program would be that the `TextIO` version would accept expressions that have no spaces, such as `17+42`, whereas the `Scanner` version would give an error for such an expression. Another is that `scanner.next()` returns a *String* rather

than a char, and the character that you want would be the first, and only, character in that string.)

The Solution

```
import textio.TextIO;

/**
 *  This program evaluates simple expressions such as 2 + 2
 *  and 34.2 * 7.81, consisting of a number, an operator,
 *  and another number.  The operators +, -, *, / are
allowed.
 *  The program will read and evaluate expressions until
 *  the user inputs a line that starts with the number 0.
 */

public class SimpleCalculator {

   public static void main(String[] args) {

      double firstNum;    // First number in the
expression.
      double secondNum;   // Second number in the
expression.
      char operator;      // The operator in the
expression.
      double value;       // The value of the expression.

      System.out.println("Enter expressions such as  2 + 2
or  34.2 * 7.81");
      System.out.println("using any of the operators +, -,
*, /.");
      System.out.println("To end, enter a 0.");
      System.out.println();

      while (true) {

          /* Get user's input, ending program if first
number is 0. */

          System.out.print("? ");
          firstNum = TextIO.getDouble();
          if (firstNum == 0)
             break;
          operator = TextIO.getChar();
```

```
          secondNum = TextIO.getlnDouble();

          /* Compute the value of the expression. */

          switch (operator) {
              case '+' -> value = firstNum + secondNum;
              case '-' -> value = firstNum - secondNum;
              case '*' -> value = firstNum * secondNum;
              case '/' -> value = firstNum / secondNum;
              default -> {
                 System.out.println("Unknown operator: " +
operator);
                 continue;  // Back to start of loop!
              }
          } // end switch

          /* Display the value. */

          System.out.println("Value is " + value);
          System.out.println();

      } // end while

      System.out.println("Good bye");

   }  // end main()

}  // end class SimpleCalculator
```

Solution for Programming Exercise 3.4

Exercise 3.4:

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.

Discussion

There are many ways to approach this problem, and probably all of them are sort of tricky to get right. Here's a simple idea that almost works: Go through

all the characters in the string. If the character is a letter, write it out. If it's not a letter, write a carriage return instead. If `line` is a *String* variable representing the line of text, this algorithm can be coded as

```
for ( i = 0;  i < line.length(); i++ ) {
   ch = line.charAt(i);
   if ( Character.isLetter(ch) )
      System.out.print(ch);
   else
      System.out.println();
}
```

This prints all the letters in a word on the same line of output. Since words in the string are separated by non-letter characters, and the computer prints a carriage return when it finds a non-letter, words in the output are separated by carriage returns. But there is a problem with this: If two words in the string are separated by **several** non-letters, then there will be one or more blank lines between the words in the output. We don't want to output two carriage returns in a row. To avoid this, we can keep track of whether the previous output was a letter or a carriage return. When we find a non-letter, we will output a carriage return only if the previous output was **not** a carriage return. To keep track of the necessary information, I'll use a boolean variable named `didCR`. The value of this variable will be true if the previous output was a carriage return. I have to remember to set the value of `didCR` each time I output something. With this modification, the code becomes:

```
for ( i = 0;  i < line.length(); i++ ) {
   ch = line.charAt(i);
   if ( Character.isLetter(ch) ) {
      System.out.print(ch);
      didCR = false;  // previous output was not a CR
   }
   else {
      if ( didCR == false ) {  // output CR, if previous
output was NOT a CR
         System.out.println();
         didCR = true;  // previous output was a CR
      }
   }
}
```

The program requires an initial value for `didCR`. In the program below, I output a carriage return before the `for` loop and set `didCR` to true. You should be able to follow the rest of the program.

An entirely different approach to this problem is given by the algorithm, "while there are more words in the string, get the next word and print it." This turns out to be even harder to implement than the above.

The Solution

```
import textio.TextIO;

/**
 *  This program will read one line of input typed by the
user.
 *  It will print the words from the input, one word to a
line.
 *  A word is defined to be a sequence of letters.  All
non-letters
 *  in the input are discarded.
 */

public class ListWordsInString {

   public static void main(String[] args) {

       String line;    // A line of text, typed in by the
user.
       int i;          // Position in line, from 0 to
line.length() - 1.
       char ch;        // One of the characters in line.
       boolean didCR;  // Set to true if the previous
output was a carriage return.

       System.out.println("Enter a line of text.");
       System.out.print("? ");
       line = TextIO.getln();

       System.out.println();
       didCR = true;

       for ( i = 0;  i < line.length();  i++ ) {
          ch = line.charAt(i);
          if ( Character.isLetter(ch) ) {
```

```
            System.out.print(ch);
            didCR = false;
         }
         else {
            if ( didCR == false ) {
               System.out.println();
               didCR = true;
            }
         }
      }

      System.out.println();  // Make sure there's at least
one carriage return at the end.

   }  // end main()

}  // end class ListWordsInString
```

Solution for Programming Exercise 3.5

Exercise 3.5:

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 2.4.4. 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.

Discussion

The statement `TextIO.readFile("sales.dat")` can be used at the beginning of the program to make *TextIO* read from the file instead of from the user's input. This statement throws an *IllegalArgumentException* if the file can't be found. Although it's not required by the statement of the problem, it would be nice to detect the exception and print an error message about the missing file. This can be done as follows, using `System.exit` to end the program if the error occurs:

```
try {
   TextIO.readFile("sales.dat");
}
catch (IllegalArgumentException e) {
   System.out.println("Can't open file \"sales.dat\" for
reading!");
   System.out.println("Please make sure the file is
present before");
   System.out.println("running the program.");
   System.exit(1);  // Terminates the program.
}
```

We have two main quantities to keep track of in the program: The total amount of sales and the number of cities for which data is missing. These quantities can be represented by two variables, `salesTotal` of type double and `missingCount` of type int. These variables are initialized to zero before processing any data from the file, and their values are output at the end of the program, after reading the entire file.

The main processing loop of the program is the `while` loop mentioned in the statement of the exercise, which begins `while (TextIO.eof() == false)`. (In my program, I've chosen to use the equivalent but more elegant test `while ( ! TextIO.eof() )`; you should try to understand why this is equivalent—and why it is more elegant!) In pseudocode form, we want a loop that does the following:

```
while there are more lines in the file:
   read the name of the city and the colon that follows it
   read the rest of the line into a string
   try to convert the string into a number
      if the conversion succeeds, add the number to the
```

```
total sales
      otherwise add 1 to the count of missing data
```

To read past the name of the city, we can simply read characters using `TextIO.getChar()` until we get to the colon:

```
do {  // Read past characters up to the ':'.
   ch = TextIO.getChar();
} while (ch != ':');
```

where `ch` is a variable of type char. We can read the rest of the line using `TextIO.getln()`. (As a side note, it's important here that we read the entire line, including the carriage return at the end. `TextIO.getln()` does this. If we neglected to read the carriage return at the end of the line, `TextIO.eof()` would not detect the end of the file correctly. After the data from the last line is read, the carriage return character would still be there in the file waiting to be read; since there is still something left to read in the file, `TextIO.eof()` would be `false` and the while loop would continue, even though all the cities have already been processed. In fact, the program would crash while looking for the next ':', which is not there. Text processing can be surprisingly subtle!)

Once we have the information in the form of a *String*, we can try to convert the string into a number with `Double.parseDouble`. If the string is not a number, this will throw an exception of type *NumberFormatException*. We can catch the error in a `try..catch` statement. The "missing data" processing is done in the `catch` part of this statement. In the program, this takes the form:

```
try {
   sales = Double.parseDouble(dataString);
   salesTotal += sales;  // This is skipped if the
conversion fails.
}
catch (NumberFormatException e) {
       // The dataString is not a number, so it counts as
       // missing data.  Add 1 to the missing data count.
   missingCount++;
}
```

(Note that my solution assumes that the data file (if it exists at all) has exactly the format that is specified in the problem. The problem would be more difficult if we had to check the format of the file, to see whether it conforms to the specification.)

The Solution

```
import textio.TextIO;

/**
 * This program reads from a file named "sales.dat".  Each
line of the
 * file contains the name of a city, followed by a colon,
followed by
 * either a number giving the amount of sales in that city
or by a
 * message saying why the sales figure is not available.
The program
 * prints the total sales for all cities and the number of
cites for
 * which the figure was not available.
 */

public class SalesFigures {

   public static void main( String[] args ) {

      /* Open file for reading; if it can't be opened, end
the program */

      try {
         TextIO.readFile("sales.dat");
      }
      catch (IllegalArgumentException e) {
         System.out.println("Can't open file \"sales.dat\"
for reading!");
         System.out.println("Please make sure the file is
present before");
         System.out.println("running the program.");
         System.exit(1);  // Terminates the program.
      }

      /* Read the file, keeping track of total sales and
missing data. */
```

```
      double salesTotal;  // Total of all sales figures
seen so far.
      int missingCount;   // Number of cities for which
data is missing.

      salesTotal = 0;
      missingCount = 0;

      while ( ! TextIO.eof() ) {  // process one line of
data.

         char ch;  // For reading past the name of the
city.
         String dataString;  // Contents of line, after the
city name.
         double sales;  // The sales figure for the city.

         do {  // Read past characters up to the ':'.
            ch = TextIO.getChar();
         } while (ch != ':');

         dataString = TextIO.getln();  // Get the rest of
the line.

         try {
            sales = Double.parseDouble(dataString);
            salesTotal += sales;  // This is skipped if the
conversion fails.
         }
         catch (NumberFormatException e) {
                // The dataString is not a number, so it
counts as
                // missing data.  Add 1 to the missing data
count.
            missingCount++;
         }

      } // end while

      /* Report the results. */

      System.out.printf("Total sales recorded from all
cities: $%1.2f\n\n", salesTotal);
      if (missingCount == 0)
         System.out.println("Data was received from all
cities.");
      else if (missingCount == 1)
```

```
         System.out.println("Data was missing from 1
city.");
      else
         System.out.printf("Data was missing from %d
cities.\n", missingCount);

   } // end main()

} // end class SalesFigures
```

Solution for Programming Exercise 3.6

Exercise 3.6:

Exercise 3.2 asked you to find the number in the range 1 to 10000 that has the largest number of divisors. You only had to print out one such number. Revise the program so that it will print out **all** numbers that have the maximum number of divisors. Use an array as follows: As you count the divisors for each number, store each count in an array. Then at the end of the program, you can go through the array and print out all the numbers that have the maximum count. The output from the program should look something like this:

```
Among integers between 1 and 10000,
The maximum number of divisors was 64
Numbers with that many divisors include:
   7560
   9240
```

Discussion

This is a fairly straightforward exercise in using arrays. We need to save 10000 numbers in an array. The numbers are the divisor counts for the numbers 1 through 10000. The numbers that we want to store in the array are of type int, so int is the base type of the array. In the program, I use an array named `saveCount` of type `int[]` to store the data. It seems natural to store the divisor count for N in array element `saveCount[N]`, but note that in that case, we don't use array element number 0, and we need an array element number 10000. Ignoring element 0 is fine, but for `saveCount[10000]` to exist, the length of the array must be at least 10001. I use an array of length 10001:

```
saveCount = new int[10001];
```

(An alternative would be to store the advisor count for N in `saveCount[N-1]`. In that case, we wouldn't need the extra array element, and the length of the array would be 10000.)

After computing each divisor count, we store the count in the array. At the same time, we are keeping track of the maximum number of divisors. After the end of the `for` loop that does the counting, we know the divisor count for each number and we know the maximum number of divisors. We just have to go through the array and print out every integer N for which the divisor count is equal to the maximum:

```
System.out.println("Numbers with that many divisors
include:");
for ( N = 1; N <= 10000; N++ ) {
    if ( saveCount[N] == maxDivisors ) {
        System.out.println( "   " + N );
    }
}
```

Note that this code checks whether the number in the array, `saveCount[N]`, is equal to the maximum, but it prints the array index, `N`, which is the integer that has that many divisors.

The Solution

New code that involves the array is shown in red italic.

```
import textio.TextIO;

/**
 * This program counts the number of divisors for integers
in the range
 * 1 to 10000.  It finds the maximum divisor count.  It
outputs the
 * maximum divisor count and a list of all integers in the
range that
 * have the maximum number of divisors.
 */

public class MostDivisorsWithArray {
```

```
   public static void main(String[] args) {

       int N;             // One of the integers whose
divisors we have to count.
       int maxDivisors;  // Maximum number of divisors seen
so far.

       int[] saveCount;  // Store the number of divisors
for each number

       saveCount = new int[10001];

       maxDivisors = 1;  // Start with the fact that 1 has
1 divisor.
       saveCount[1] = 1;

       /* Process all the remaining values of N from 2 to
10000, and store
          all the divisor counts in the array.  Update the
value of maxDivisor
          whenever we find a value of N that has more
divisors than the current
          value.
       */

       for ( N = 2;  N <= 10000;  N++ ) {

           int D;  // A number to be tested to see if it's
a divisor of N.
           int divisorCount;  // Number of divisors of N.

           divisorCount = 0;

           for ( D = 1;  D <= N;  D++ ) {  // Count the
divisors of N.
              if ( N % D == 0 )
                 divisorCount++;
           }

           saveCount[N] = divisorCount;  // Record the
count for N in the array

           if (divisorCount > maxDivisors) {
              maxDivisors = divisorCount;
           }

       }
```

```
        System.out.println("Among integers between 1 and
10000,");
        System.out.println("The maximum number of divisors
was " + maxDivisors);
        System.out.println("Numbers with that many divisors
include:");
        for ( N = 1; N <= 10000; N++ ) {
            if ( saveCount[N] == maxDivisors ) {
                System.out.println( "   " + N );
            }
        }

    } // end main()

}
```

Solution for Programming Exercise 3.7

Exercise 3.7:

An example in Subsection 3.8.3 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 BirthdayProblem.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.)

Discussion

The original program and the three programs for this exercise have some similarities, and we will use ideas from the original program. However, each part of this exercise presents its own problem.

The first question asks how many people you have to choose at random before finding three who share the same birthday. This is similar to the original program, but instead of just checking whether or not a given birthday has already been found, we need to keep track of how many people have

been found with each birthday. Where the original program used an array of `booleans`, here we need an array of `ints`. We still want to count the number of people we check and output that count at the end. An algorithm for the simulation is:

```
Let count = 0
Repeat:
   Select a birthday at random
   Add one to count
   If this is the third time that birthday has occurred:
       break out of the loop
   Add one to the number of times that birthday has been
found
Output the count
```

The original program used a boolean array to keep track of whether or not each day had been seen. For this proble, we need to know *how many times* each day has been seen. That means that instead of one boolean value for each day, we need one integer value for each day. So, to do the counting, we need an array "`int[] numFound`", where `numFound[i]` will be the number of times the `i`-th day of the year has occurred as a birthday. Since `numFound[i]` can be used in any way that any int variable can be used, we can add one to the number stored in `numFound[i]` by saying "`numFound[i]++`" or "`numFound[i] = numFound[i] + 1`". When we create the array with the command "`numFound = new int[365]`", all the elements of the array are automatically initialized to zero. This is the initial value that we want. (This is an important thing to remember! In some programming languages, arrays are not automatically filled with zeros, so it would be necessary to use a `for` loop to store a zero into each place in the array. Even in Java, if you reuse the same array rather than creating a new one for each use, you would have to remember to set initialize each element of the array before reusing it.)

Given all this, we can translate the algorithm into Java as follows:

```
int[] numFound;  // numFound[i] will be the number of
people
                 // who have been found who have a
birthday
                 // on the i-th day of the year
```

```
int count;        // The number of people who have been
checked.

numFound = new int[365];  // Initially, all entries are 0.

count = 0;

while (true) {
       // Select a birthday at random, from 0 to 364.
       // If the same birthday was already seen twice
       // before, then quit.  Otherwise, add one to
       // the corresponding entry in the numFound array
       // to record that a person with that birthday
       // has been found.
   int birthday;  // The selected birthday.
   birthday = (int)(Math.random()*365);
   count++;
   if ( numFound[birthday] == 2 )
       break;  // It's the third time we've found this
birthday.
   numFound[birthday]++;
}

System.out.println("It took " + count +
        " tries to find three people with the same
birthday.");
```

The lines shown in red italic are the ones that differ significantly from the original program. This becomes the body of the `main()` subroutine in the first program.

For the second program, we know exactly how many people we want to check: 365. This calls for using a for loop. The information we need for each birthday is whether or not that birthday has occurred. For that, we can use an array of booleans. After the `for` loop, value stored in position `i` of the array will `true` if the `i`-th day of the year has occurred as a birthday and is `false` if not. After checking 365 people, we have to go through the boolean array and count the number of entries in the array that are `true`. This is the number of different birthdays that have been found. The algorithm can be expressed as:

```
Let used = new boolean[365]
Repeat 365 times:
   Select a birthday at random
   Store true into the corresponding location in the
array, used
Let count = 0
for day = 0 to 364:
   If used[day] is true:
      Add 1 to count
Output the value of count
```

This translates easily into Java code:

```
boolean used[];  // used[i] will be true if a person is
found
                 // whose birthday is the i-th day of the
year.

used = new boolean[365];  // Initially, all entries are
false!

for (i = 0; i < 365; i++) {
      // Select a random birthday and record it.
   int birthday;  // The selected birthday.
   birthday = (int)(Math.random()*365);
   used[birthday] = true;
}

int count = 0;

for (day = 0; day < 365; day++) {
       // If this day occurred as a birthday, count it.
   if (used[day] == true)
      count++;
}

System.out.println("Among 365 people, there were " + count
                                + " different
birthdays.");
```

It might be worth noting that the test "`if (used[day] == true)`" can be written more simply—and elegantly—as "`if (used[day])`". Also, the three lines in the first `for` loop could be reduced to the single command "`used[(int)(Math.random()*365)] = true;`". Of course, this one-line version is harder to understand.

The third program is just a little bit trickier. We have to continue selecting people at random until we have found 365 different birthdays. We can use a counter to keep track of how many different birthdays we have found. We have to continue until this counter reaches 365. We need a second counter to keep track of how many different people we have checked. It's the second counter whose value we want to output at the end. Now, we have to be able to recognize whether a birthday we've just found is new, or whether we've encountered it previously. For this, we can again use an array of booleans. An algorithm for the simulation is:

```
Let used = new boolean[365]
Let count = 0   // The number of people checked
Let birthdaysFound = 0   // The number of different
birthdays found
while birthdaysFound < 365:
   Add one to count
   Select a birthday at random
   if used[birthday] is false:
       Add one to birthdaysFound  // since this is a new
birthday
   Let used[birthday] = true   // so we don't count it
again
Output the value of count
```

In Java, this algorithm becomes:

```
boolean[] used;  // For recording the possible birthdays
                 //   that have been seen so far.
int count;       // The number of people who have been
checked.
int birthdaysFound;   // The number of different birthdays
that
                      // have been found.

used = new boolean[365];  // Initially, all entries are
false.
count = 0;
birthdaysFound = 0;

while (birthdaysFound < 365) {
       // Select a birthday at random, from 0 to 364.
       // If the birthday has not already been used,
       // add 1 to birthdaysFound.
```

```
    int birthday;  // The selected birthday.
    birthday = (int)(Math.random()*365);
    count++;
    if ( used[birthday] == false )
       birthdaysFound++;
    used[birthday] = true;
 }

 System.out.println( count + " people were checked." );
```

The Solution

Finding three people with the same birthday:

```
/**
 *  How many random people do you have to select before you
find
 *  THREE with the same birthday (that is, three people who
were born
 *  on the same day of the same month, but not necessarily
in the
 *  same year).  This program simulates the process.  (It
ignores the
 *  possibility of people born on leap day.)
 */

public class BirthdayProblem2 {

   /**
    * Simulate choosing people at random and checking the
    * day of the year they were born on.  If the person is
    * the third who was born on that day of the year, stop,
    * and output the number of people who were checked.
    */
   public static void main(String[] args) {

      int[] numFound;  // numFound[i] will be the number of
people
                       // who have been found who have a
birthday
                       // on the i-th day of the year

      int count;       // The number of people who have
been checked.
```

```
      numFound = new int[365];  // Initially, all entries
are 0.

      count = 0;

      while (true) {
             // Select a birthday at random, from 0 to 364.
             // If the same birthday was already seen twice
             // before, then quit.  Otherwise, add one to
             // the corresponding entry in the numFound
array
             // to record that a person with that birthday
             // has been found.
         int birthday;  // The selected birthday.
         birthday = (int)(Math.random()*365);
         count++;
         if ( numFound[birthday] == 2 )
            break;
         numFound[birthday]++;
      }

      System.out.println("It took " + count +
               " tries to find three people with the same
birthday.");

   }


} // end class BirthdayProblem2
```

How many different birthdays do 365 people have?

```
/**
 * This program simulates selecting 365 people at random
and finding
 * how many different birthdays they have among them.
 */

public class BirthdayProblem3 {

   /**
    * Simulate choosing people at random and checking the
    * day of the year they were born on.  The number of
    * different days found among 365 people is counted
    * and output.
    */
```

```
   public static void main(String[] args) {

      boolean used[];  // used[i] will be true if a person
is found
                       // whose birthday is on the i-th day
of the year.

      used = new boolean[365];  // Initially, all entries
are false.

      /* Choose 365 days at random, and mark each day by
         setting the corresponding entry in the array,
used,
         to true.  (If the value is already true, it
doesn't
         matter.  We are only interested in whether or not
         the birthday occurs, not how many times it
occurs.)
      */

      int i;
      for (i = 0; i < 365; i++) {
         int birthday;  // The selected birthday.
         birthday = (int)(Math.random()*365);
         used[birthday] = true;
      }

      /* Now, count how many entries in the used array are
true.
         This is how many different birthdays were found.
      */

      int count = 0;
      int day;

      for (day = 0; day < 365; day++) {
             // If this day occurred as a birthday, count
it.
         if (used[day] == true)
            count++;
      }

      System.out.println("Among 365 people, there were " +
count
                                        + " different
birthdays.");

   }
```

```
} // end class BirthdayProblem3
```

Finding 365 different birthdays:

```
/**
 * How many random people do you have to select before you
 *  have found someone with every possible birthday
(ignoring
 *  leap years)?  This program simulates the process.
 */

public class BirthdayProblem4 {

    /**
     * Simulate choosing people at random and checking the
     * day of the year they were born on.  People are
chosen
     * until all 365 possible birthdays (ignoring leap
years)
     * have been found.  Then the number of people surveyed
     * is output.
     */
   public static void main(String[] args) {

       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.

       int birthdaysFound;   // The number of different
birthdays that
                             // have been found.

       used = new boolean[365];  // Initially, all entries
are false.
```

```
        count = 0;

        birthdaysFound = 0;

        while (birthdaysFound < 365) {
               // Select a birthday at random, from 0 to
364.
               // If the birthday has not already been used,
               // add 1 to birthdaysFound.
           int birthday;  // The selected birthday.
           birthday = (int)(Math.random()*365);
           count++;
           if ( used[birthday] == false )
              birthdaysFound++;
           used[birthday] = true;
        }

        System.out.println( count + " people were checked."
);

    }


} // end class BirthdayProblem4
```

Solution for Programming Exercise 3.8

Exercise 3.8:

Write a GUI program that draws a checkerboard. Base your solution on the sample program SimpleGraphicsStarter.java You will draw the checkerboard in the `drawPicture()` subroutine, after erasing the code that it already contains.

The checkerboard should be 400-by-400 pixels. You can change the size of the drawing area in `SimpleGraphicsStarter.java` by modifying the first two lines of the `start()` subroutine to set `width` and `height` to 400 instead of 800 and 600. A checkerboard contains 8 rows and 8 columns of squares. If the size of the drawing area is 400, that means that each square should be 50-by-50 pixels. The squares are red and black (or whatever other colors you choose). Here is a tricky way to determine whether a given square should be red or black: The rows and columns can be thought of as numbered from 0 to 7. If the row number of the square and the column number of the square 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 a reduced-size image of the checkerboard that you want to draw:

Discussion

The basic algorithm is obvious:

```
for each row of the checkerboard:
    Draw all the squares in that row
```

Since any given row contains eight squares, one in each column of the checkerboard, we can expand the body of the for loop into another for loop:

```
for each of the eight rows of the checkerboard:
    for each of the eight columns:
        Draw the square in that row and column
```

Each square is a rectangle with height 50 and width 50, so it can be drawn with the command `g.fillRect(x,y,50,50)`, where `x` and `y` are the coordinates of the top-left corner of the square. Before drawing the square, we have to determine whether it should be red or black, and we have to set the correct color with `g.setFill`. So, the algorithm becomes

```
for each row on the checkerboard:
    for each of the eight columns:
        Compute x,y for the top-left corner of the square
        if it's a red square:
           g.setFill(Color.RED)
        else
           g.setFill(Color.BLACK)
        g.fillRect(x,y,50,50)
```

The top of the first row of squares is at `y=0`. Since each square is 50 pixels high, the top of the second row is at `y=50`, followed by 100 for the third row, then 150, 200, 250, 300, and 350. If we assume that the rows are numbered 0, 1, 2, ..., 7, then the tops are given by `y = row*50`, where `row` is the row number. (If you number the rows 1, 2, ..., 8, the formula would be `(row-1)*20`. The simpler formula in this and in many similar cases is one reason why computer scientists like to start counting with 0 instead of 1.) Similarly, the left edge of the squares in column `col` is given by `x = col*50`, where again the columns are numbered 0, 1, 2, ..., 7. I'll use "`for (row=0; row<8; row++)`" to count off the rows, rather than the equivalent "`for (row=0; row<=7; row++)`". The 8 reminds me that I am counting off the **eight** numbers 0, 1, 2, ..., 7. Again, this is typical computer science style.

The only problem remaining is how to determine whether the square is red. As noted in the exercise, a square is red if `row` and `col` are either both even

or both odd. Since an integer `N` is even if `N%2` is 0, the test could be expressed as

```
if ((row%2 == 0 && col%2 == 0) || (row%2 == 1 && col%2 ==
1))
```

However, note that this is the same as asking whether `row%2` and `col%2` have the same value. So the test can be written more simply as "`if (row%2 == col%2)`". Putting this all together into syntactically correct Java code, the algorithm becomes

```
for ( row = 0;  row < 8;  row++ ) {
   for ( col = 0;  col < 8;  col++ ) {
       x = 50*col;
       y = 50*row;
       if ( (row % 2) == (col % 2) )
          g.setFill(Color.RED);
       else
          g.setFill(Color.BLACK);
       g.fillRect(x,y,50,50);
   }
}
```

Of course, the variables `row`, `col`, `x`, and `y` have to be declared to be of type int. Then, the code goes into the body of the `drawPicture()` subroutine.

The complete program is shown below. In the original program, the size of the drawing area was 800-by-600. I changed this to 400-by-400, as noted in the exercise. And I changed the title of the window to "Checkerboard" in another line in `start()`.

I also changed the name of the class from *SimpleGraphicsStarter* to *Checkerboard*. Remember that this also requires changing the name of the file from `SimpleGraphicsStarter.java` to `Checkerboard.java`.

The Solution

```
import javafx.application.Application;
import javafx.scene.layout.BorderPane;
import javafx.scene.Scene;
import javafx.stage.Stage;
```

```
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.paint.Color;

/**
 *  This program draws a checkerboard
 */
public class Checkerboard extends Application {

    /**
     * Draws a picture.  The parameters width and height
give the size
     * of the drawing area, in pixels.
     */
    public void drawPicture(GraphicsContext g, int width,
int height) {

          int row;   // Row number, from 0 to 7
          int col;   // Column number, from 0 to 7
          int x,y;   // Top-left corner of square

          for ( row = 0;  row < 8;  row++ ) {

             for ( col = 0;  col < 8;  col++) {
                x = col * 50;
                y = row * 50;
                if ( (row % 2) == (col % 2) )
                   g.setFill(Color.RED);
                else
                   g.setFill(Color.BLACK);
                g.fillRect(x, y, 50, 50);
             }

          }

    } // end drawPicture()

    //------ Implementation details: DO NOT EXPECT TO
UNDERSTAND THIS ------


    public void start(Stage stage) {
        int width = 400;   // The width of the image.  You
can modify this value!
        int height = 400;  // The height of the image. You
can modify this value!
        Canvas canvas = new Canvas(width,height);
        drawPicture(canvas.getGraphicsContext2D(), width,
```

```
height);
        BorderPane root = new BorderPane(canvas);
        root.setStyle("-fx-border-width: 4px; -fx-border-
color: #444");
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Checkerboard"); // STRING APPEARS
IN WINDOW TITLEBAR!
        stage.show();
        stage.setResizable(false);
    }

    public static void main(String[] args) {
        launch();
    }

} // end Checkerboard
```

Solution for Programming Exercise 3.9

Exercise 3.9:

Often, some element of an animation repeats over and over, every so many frames. Sometimes, the repetition is "cyclic," meaning that at the end it jumps back to the start. Sometimes the repetition is "oscillating," like a back-and-forth motion where the second half is the same as the first half played in reverse.

Write an animation that demonstrates both cyclic and oscillating motions at various speeds. For cyclic motion, you can use a square that moves across the drawing area, then jumps back to the start, and then repeats the same motion over and over. For oscillating motion, you can do something similar, but the square should move back and forth between the two edges of the drawing area; that is, it moves left-to-right during the first half of the animation and then backwards from right-to-left during the second half. To write the program, you can start with a copy of the sample program SimpleAnimationStarter.java.

A cyclic motion has to repeat every N frames for some value of N. What you draw in some frame of the animation depends on the `frameNumber`. The `frameNumber` just keeps increasing forever. To implement cyclic motion, what you really want is a "cyclic frame number" that takes on the values 0, 1, 2, ..., (N-1), 0, 1, 2, ..., (N-1), 0, 1, 2, ... You can derive the value that you need from `frameNumber` simply by saying

```
cyclicFrameNumber = frameNumber % N;
```

Then, you just have to base what you draw on `cyclicFrameNumber` instead of on `frameNumber`. Similarly, for an oscillating animation, you need an "oscillation frame number" that takes on the values 0, 1, 2, ... (N-1), N, (N-1), (N-2), ... 2, 1, 0, 1, 2, and so on, repeating the back and forth motion forever. You can compute the value that you need with

```
oscilationFrameNumber = frameNumber % (2*N);
if (oscillationFrameNumber > N)
   oscillationFrameNumber = (2*N) -
oscillationFrameNumber;
```

Here is a screen shot from my version of the program. I use six squares. The top three do cyclic motion at various speeds, while the bottom three do oscillating motion. I drew black lines across the drawing area to separate the squares and to give them "channels" to move in.

Discussion

This exercise exists mostly to introduce you to cyclic and oscillating motion and the formulas that are used to implement them. But it's also a good exercise in designing an animation and working with coordinates.

Some decisions have to be made before you can start writing code. I decided to use rather small squares. (Actually, I tried it with big squares and didn't like the appearance as much.) The size of my squares is 20-by-20 pixels. For my first cyclic motion, I decided to make the length of the animation 300 frames, or about 5 seconds. This means that `cyclicFrameNumber` can be computed as `frameNumber % 300`. And I decided to move the square at one pixel per frame; that lets me use `cyclicFrameNumber` as the x-coordinate for the square. The top square moves along the top edge of the window, so its y-coordinate is 0. The code for the first square is:

```
cyclicFrameNum = frameNumber % 300;  // Repeats every 300
frames
g.setFill(Color.RED);
g.fillRect( cyclicFrameNum, 0, 20, 20 );
```

The exercise asks for cyclic motion at different speeds. If the second square is to be faster than the first, it should cross the drawing area in a smaller

number of frames. If it is twice as fast, it will only need half as many frames to make it across the drawing area. For the second square, I use a cycle length of 150 instead of 300. To cross the entire area, the x-coordinate of the square must get to 300 after just 150 frames. We can accomplish that by using `2*cyclicFrameNumber` as the x-coord. (Another way to think about it: A square with x-coord `2*cyclicFrameNumber` moves twice as fast as one with x-coord `cyclicFrameNumber`. And it will finish the journey after 150 frames instead of 300, so the length of the cycle should be 150 frames.) The code for the second square is

```
cyclicFrameNum = frameNumber % 150;  // Repeats every 150
frames
g.setFill(Color.GREEN);
g.fillRect( 2*cyclicFrameNum, 20, 20, 20 );
```

The y-coord is 20, since the top square occupies 20 pixels at top of the window. The third square moves three times as fast as the first, and its code is similar.

For the fourth square, I want an oscillating motion. Like the first square, it takes 300 frames to move across the window from left to right. But then it takes another 300 frames to move back from right to left. The total length of the motion is 600. (In the formula given in the exercise, N is 300 and 2*N is 600.) The formula for the oscillation frame number is

```
oscilationFrameNum = frameNumber % 600;
if (oscilationFrameNum > 300)
    oscilationFrameNum = 600 - oscilationFrameNum;
```

You should study how this works. When `oscilationFrameNumber` is between 300 and 600, the formula `600-oscilationFrameNumber` gives an answer between 0 and 300. And as `oscilationFrameNumber` gets bigger, `600-oscilationFrameNumber` gets smaller (which will mean that the x-coord of the square is decreasing). From there, you should be able to figure out the rest of the program.

One more small issue is the size of the drawing area. Since there are 6 squares, and each is 20 pixels high, the height of the drawing area must be 120. You might think at first that the width should be 300, since the x-

coordinate of the square goes up to 300. But that x-coordinate is actually where the left edge of the square is. When the left edge is at 300, the right edge is at 320. So for the entire square to be visible, we have to make the width 320.

For drawing the horizontal black lines, the program uses the statement

```
g.strokeLine(0,y+0.5,320,y+0.5);
```

(I should explain the "+ 0.5". The coordinates in `g.strokeLine()` give the position of the **centers** of the endpoints of the line. Without the "+ "0.5", the top half of the line drawn here would lie in one row of pixels, and the bottom half would lie in the next row. The pixels in each row are only half covered by the line! The computer might, depending on the environment, color a partially covered pixel by blending the drawing color with the color of the background, so in this case, we would actually get a two pixel wide gray line instead of a one pixel wide black line. This is an example of something called "antialiasing," which you will encounter again later in the text. By moving the line down 0.5 pixels, it will exactly cover one row of pixels, which will be filled with solid black.)

The Solution

```
import javafx.animation.AnimationTimer;
import javafx.application.Application;
import javafx.scene.layout.BorderPane;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.paint.Color;

/**
 *  This file can be used to create very simple animations.
Just fill in
 *  the definition of drawFrame with the code to draw one
frame of the
 *  animation, and possibly change a few of the values in
the rest of
 *  the program as noted below.
 */
```

```
public class CyclicAndOscillatingMotionDemo extends
Application {

    /**
     * Draws one frame of an animation. This subroutine
should be called
     * about 60 times per second.  It is responsible for
redrawing the
     * entire drawing area. The parameter g is used for
drawing. The frameNumber
     * starts at zero and increases by 1 each time this
subroutine is called.
     * The parameter elapsedSeconds gives the number of
seconds since the animation
     * was started.  By using frameNumber and/or
elapsedSeconds in the drawing
     * code, you can make a picture that changes over time.
That's an animation.
     * The parameters width and height give the size of the
drawing area, in pixels.
     */
    public void drawFrame(GraphicsContext g, int
frameNumber, double elapsedSeconds, int width, int height)
{

        g.setFill(Color.WHITE);
        g.fillRect(0, 0, width, height); // First, fill the
entire image with a background color!

        /* Show cyclic motion at three speeds.  In each
case, a square
         * moves across the drawing area from left to
right, then jumps
         * back to the start.
         */

        int cyclicFrameNum;

        cyclicFrameNum = frameNumber % 300;  // Repeats
every 300 frames
        g.setFill(Color.RED);
        g.fillRect( cyclicFrameNum, 0, 20, 20 );

        cyclicFrameNum = frameNumber % 150;  // Repeats
every 150 frames
        g.setFill(Color.GREEN);
        g.fillRect( 2*cyclicFrameNum, 20, 20, 20 );
```

```
        cyclicFrameNum = frameNumber % 100;  // Repeats
every 100 frames
        g.setFill(Color.BLUE);
        g.fillRect( 3*cyclicFrameNum, 40, 20, 20 );


        /* Show oscillating motion at three speeds.  In
each case, a square
         * moves across the drawing area from left to
right, then reverses
         * direction to move from right to left back to its
starting point.
         */

        int oscillationFrameNum;

        oscillationFrameNum = frameNumber % 600;  //
repeats every 600 frames
        if (oscillationFrameNum > 300)
            oscillationFrameNum = 600 -
oscillationFrameNum; // after 300, the values go backwards
back to 0
        g.setFill(Color.CYAN);
        g.fillRect( oscillationFrameNum, 60, 20, 20 );

        oscillationFrameNum = frameNumber % 300; // repeats
every 300 frames
        if (oscillationFrameNum > 150)
            oscillationFrameNum = 300 -
oscillationFrameNum; // after 150, the values go backwards
back to 0
        g.setFill(Color.MAGENTA);
        g.fillRect( 2*oscillationFrameNum, 80, 20, 20 );

        oscillationFrameNum = frameNumber % 200; // repeats
every 200 frames
        if (oscillationFrameNum > 100)
            oscillationFrameNum = 200 -
oscillationFrameNum; // after 100, the values go backwards
back to 0
        g.setFill(Color.YELLOW);
        g.fillRect( 3*oscillationFrameNum, 100, 20, 20 );


        /* Draw horizontal black lines across the window to
separate the
         * regions used by the six squares.  Also draw a
box around the outside,
```

```
         * mostly for the picture that I need for the web
page!
         */

        int y;
        g.setStroke(Color.BLACK);
        for ( y = 20; y < 120; y = y + 20 )
            g.strokeLine(0,y+0.5,320,y+0.5);
     }

    //------ Implementation details: DO NOT EXPECT TO
UNDERSTAND THIS ------


    public void start(Stage stage) {
        int width = 320;   // The width of the image.  You
can modify this value!
        int height = 120;  // The height of the image. You
can modify this value!
        Canvas canvas = new Canvas(width,height);
        drawFrame(canvas.getGraphicsContext2D(), 0, 0,
width, height);
        BorderPane root = new BorderPane(canvas);
        root.setStyle("-fx-border-width: 4px; -fx-border-
color: #444");
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Motion demo"); // STRING APPEARS IN
WINDOW TITLEBAR!
        stage.show();
        stage.setResizable(false);
        AnimationTimer anim = new AnimationTimer() {
            private int frameNum;
            private long startTime = -1;
            private long previousTime;
            public void handle(long now) {
                if (startTime < 0) {
                    startTime = previousTime = now;

drawFrame(canvas.getGraphicsContext2D(), 0, 0, width,
height);
                }
                else if (now - previousTime > 0.95e9/60) {
                       // The test in the else-if is to
guard against a bug that has shown
                       // up in some versions of JavaFX on
some computers.  The bug allows
                       // the handle() method to be called
```

```
many times more than the 60 times
                        // per second that is specified in
the JavaFX documentation.
                    frameNum++;

drawFrame(canvas.getGraphicsContext2D(), frameNum, (now-
startTime)/1e9, width, height);
                    previousTime = now;
                }
            }
        };
        anim.start();
    }

    public static void main(String[] args) {
        launch();
    }

} // end CyclicAndOscillatingMotionDemo
```

Solution for Programming Exercise 4.1

Exercise 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 from Exercise 3.4 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.

Discussion

We are told the name of the subroutine and that it has one parameter of type *String*. The name of the parameter is not specified. I will use `str`. The return type is `void` because the subroutine does not return a value. (It displays a value to the user, but to **return** a value means to return it to the line in the program where the function is called. The value returned by a function is generally **not** displayed to the user by the function.) The first line of the subroutine definition will be:

```
static void printCapitalized( String str )
```

Adding a `public` or `private` access modifer at the beginning of the line is optional. Note that `static` is not optional, since this subroutine will be used in the `static` subroutine `main()`, and it is not legal for a `static` routine to use a non-`static` subroutine.

As for the code inside the subroutine definition, it must look at each character in `str` and decide whether to capitalize it or not. An algorithm for the subroutine is

```
for each character in str:
    if the character is the first letter of a word:
        Print a capitalized version of the character
    else:
        Print the character
Print a carriage return to end the line of output
```

The test as to whether a character is the first letter of a word is surprisingly complicated. A test that almost works is: "If the character is a letter and the preceding character is not a letter." The problem is that if the character is the first character is the string, then there is no preceding character! If the character is `str.charAt(i)`, then the preceding character would be `str.charAt(i-1)`, but `str.charAt(i-1)` doesn't exist if `i` is `0`. You should notice the problem when you look at `str.charAt(i-1)` and remember that the `charAt()` method has the precondition that its parameter must be greater than or equal to zero. When `i` is zero in `str.charAt(i-1)`, that precondition is not satisfied.

Let's look at Java code that suffers from this bug. Recall that the operator "`!`" stands for "not."

```
for ( i = 0;  i < str.length();  i++ ) {        // BUGGY
CODE!
    ch = str.charAt( i );
    if ( Character.isLetter(ch) && !
Character.isLetter(str.charAt(i-1)) )
        System.out.print( Character.toUpperCase(ch) );
    else
        System.out.print( ch );
}
System.out.println();
```

This will crash when `i` is zero, if the first character in the string is indeed a letter. There are several ways to work around the problem, and all of them are techniques that are worth knowing. The first is to use a more complicated test in the `if` statement: "if the character is a letter and either it's the first character in the string or the previous character is not a letter". In Java, this is:

```
if (Character.isLetter(ch) && (i==0 || !
Character.isLetter(str.charAt(i-1))))
```

This avoids testing `str.charAt(i-1)` in the case when `i` is zero. But it can be difficult to get such a complicated test right. Another possibility is a bit sneaky: Add an extra character onto the beginning of `str`, and then start the `for` loop with `i=1`. Any character will do, as long as it's not a letter. For example, you could say "`str = "." + str;`" Since the `for` loop starts at `i=1`, the "." is not copied to output, and the problem of `i == 0` doesn't arise. The method that I will use is similar, but it doesn't require any modification of `str`. I'll use another variable to represent the preceding character in the string, except that at the beginning of the string, I'll set it to the arbitrary value, '.'. At the end of the loop, the character that we have just processed becomes the "previous character" in the next iteration of the loop. Here is the complete subroutine, using this method:

```
static void printCapitalized( String str ) {
   char ch;        // One of the characters in str.
   char prevCh;    // The character that comes before ch in
the string.
   int i;          // A position in str, from 0 to
str.length()-1.
   prevCh = '.';   // Prime the loop with any non-letter
character.
   for ( i = 0;  i < str.length();  i++ ) {
      ch = str.charAt(i);
      if ( Character.isLetter(ch)  &&  !
Character.isLetter(prevCh) )
          System.out.print( Character.toUpperCase(ch) );
      else
          System.out.print( ch );
      prevCh = ch;  // prevCh for next iteration is ch.
   }
   System.out.println();
}
```

Keeping track of a previous value in a loop is a very common programming pattern. This doesn't exhaust the possibilities. Another idea, for example, would be to use a boolean variable to keep track of whether the previous character was a letter.

Finally, we should add a Javadoc comment to document the subroutine. Writing a `main()` routine to test this subroutine on a line of input is easy.

The Solution

```
import textio.TextIO;

/**
 * This program will get a line of input from the user and
will print a copy
 * of the line in which the first character of each word
has been changed to
 * upper case.  The program was written to test the
printCapitalized
 * subroutine.  It depends on the non-standard TextIO
class.
 */

public class CapitalizeOneString {

   public static void main(String[] args) {
      String line;  // Line of text entered by user.
      System.out.println("Enter a line of text.");
      line = TextIO.getln();
      System.out.println();
      System.out.println("Capitalized version:");
      printCapitalized( line );
   }

   /**
    *  Print a copy of a string to standard output, with
the first letter
    *  of each word in the string changed to upper case.
    *  @param str the string that is to be output in
capitalized form
    */
   static void printCapitalized( String str ) {
      char ch;       // One of the characters in str.
      char prevCh;   // The character that comes before ch
```

```
in the string.
      int i;         // A position in str, from 0 to
str.length()-1.
      prevCh = '.';  // Prime the loop with any non-letter
character.
      for ( i = 0;  i < str.length();  i++ ) {
         ch = str.charAt(i);
         if ( Character.isLetter(ch)  &&  !
Character.isLetter(prevCh) )
            System.out.print( Character.toUpperCase(ch) );
         else
            System.out.print( ch );
         prevCh = ch;  // prevCh for next iteration is ch.
      }
      System.out.println();
   }

} // end CapitalizeOneString
```

Solution for Programming Exercise 4.2

Exercise 4.2:

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.

Discussion

The subroutine has a parameter of type char and a return value of type int. It's easy to write the `switch` statement, although it's tedious because of the number of cases. A little creative cut-and-paste can help. The `switch` statement has a `default` case that covers all the characters that are not hexadecimal digits. For such characters, a value of `-1` is returned. Note that

the action in each case of the `switch` is to return a value. A `return` statement is not allowed after the `->` in a case (only subroutine call statements, `throw` statements, and block statements are allowed). So the `return` statements in the example are enclosed between braces to make it into a block statement. For example,

```
case 0 -> { return 0; }
```

The complete subroutine is shown in the program below. It would actually make more sense in this program to use a `switch` expression (Subsection 3.6.5) to compute the return value. Then the suboutine could be written:

```
public static int hexValue(char ch) {
    return  switch(ch) {
       case '0' -> 0;
       case '1' -> 1;
       case '2' -> 2;
       case '3' -> 3;
       case '4' -> 4;
       case '5' -> 5;
       case '6' -> 6;
       case '7' -> 7;
       case '8' -> 8;
       case '9' -> 9;
       case 'A', 'a' -> 10;
       case 'B', 'b' -> 11;
       case 'C', 'c' -> 12;
       case 'D', 'd' -> 13;
       case 'E', 'e' -> 14;
       case 'F', 'f' -> 15;
       default  -> -1;
    };
}
```

In the main program, I will use `TextIO.getlnWord()` to read the user's input, rather than `TextIO.getln()`. This has the advantage that it will return a non-empty string that is guaranteed not to contain any blanks. We still have the problem of checking whether the user's input contains only valid hexadecimal digits. One approach is to check all the characters first and use a boolean variable to record whether they are all valid. We can test whether

an individual character, `ch`, is valid by checking whether `hexValue(ch)` is `-1`. Let `hex` be a *String* holding the user's input. Then we have:

```
boolean valid; // Flag variable for testing whether the
string is valid.
valid = true;  // Assume that the input is valid, and
change our
               // mind if we find an invalid character.
for ( i = 0; i < hex.length(); i++ ) {
    if ( hexValue(hex.charAt(i)) == -1 ) {  // Character
number i is bad.
       valid = false;
       break; // Leave the for loop, since we are now sure
of the answer.
    }
}

// At the point, valid is true if and only if we have
examined all the
// characters in the string and found that each one is a
hex digit.

if ( valid ) {  // If the input is valid, compute and
print base-10 value
   dec = 0;
   for ( i = 0; i < hex.length(); i++ )
      dec = 16*dec + hexValue( hex.charAt(i) );
   System.out.println("Base-10 value is:  " + dec);
}
else {  // Input is not valid, print an error message
   System.out.println("Error:  Input is not a hexadecimal
number.");
}
```

This works, but we have to process the string twice. We can avoid this by checking the input at the same time that we do the conversion. If the input is illegal, I want to end the program. I use the fact that a `return` statement in the `main()` routine will end the program, since it returns control back to the system:

```
dec = 0;
for ( i = 0; i < hex.length(); i++) {
   int digit = hexValue( hex.charAt(i) );
   if (digit == -1) {
       System.out.println("Error:  Input is not a
```

```
hexadecimal number.");
        return;  // Ends the main() routine.
    }
    dec = 16*dec + digit;
}
System.out.println("Base-10 value:  " + dec);
```

This is the code that is used in the `main()` routine of the program to do the conversion. Note that I declared `dec` to be of type long to allow bigger values than would fit in a variable of type int. The program still has a problem if the user enters too many characters. (It gets the wrong answer.)

It would probably be better to write a function to do the conversion of a string to base-10. It could return a `-1` if the string is not a legal hexadecimal number. The `main()` routine could then call the function and check its return value to find out whether there was an error.

In this exercise, a special value, `-1`, is returned by a subroutine to indicate that its parameter value is not valid. An alternative approach would be to throw an exception if the parameter value is not legal. Instead of saying "`return -1`", the `default` case of the `switch` statement in the subroutine could read:

```
default:
   throw new IllegalArgumentException("Not a legal
hexadecimal digit: '" + ch + "'.");
```

The `main` program would then do the conversion from string to hexadecimal in a `try..catch` statement. If an exception occurs, the `catch` part of the statement can print an error message:

```
try {    // IF WE WERE USING AN EXCEPTION-THROWING VERSION
OF hexValue
    dec = 0;
    for ( i = 0; i < hex.length(); i++ ) {
       int digit = hexValue( hex.charAt(i) );  // Might
throw an exception.
       dec = 16*dec + digit;
    }
    // We get here only if no exception occurred.
    System.out.println("Base-10 value:  " + dec);
```

```
}
catch ( IllegalArgumentException e ) {
   System.out.println("Error:  Input is not a hexadecimal
number.");
}
```

I think that this approach is much neater!

The Solution

```
import textio.TextIO;

/**
 * This program reads a hexadecimal number input by the
user and prints the
 * base-10 equivalent.  If the input contains characters
that are not
 * hexadecimal numbers, then an error message is printed.
 */

public class Hex2Dec {

    public static void main(String[] args) {
       String hex;  // Input from user, containing a
hexadecimal number.
       long dec;    // Decimal (base-10) equivalent of
hexadecimal number.
       int i;       // A position in hex, from 0 to
hex.length()-1.
       System.out.print("Enter a hexadecimal number: ");
       hex = TextIO.getlnWord();
       dec = 0;
       for ( i = 0; i < hex.length(); i++ ) {
          int digit = hexValue( hex.charAt(i) );
          if (digit == -1) {
              System.out.println("Error:  Input is not a
hexadecimal number.");
              return;  // Ends the main() routine.
          }
          dec = 16*dec + digit;
       }
       System.out.println("Base-10 value:  " + dec);
    }  // end main

    /**
```

```
     * Returns the hexadecimal value of a given character,
or -1 if it is not
     * a valid hexadecimal digit.
     * @param ch the character that is to be converted into
a hexadecimal digit
     * @return the hexadecimal value of ch, or -1 if ch is
not
     *     a legal hexadecimal digit
     */
    public static int hexValue(char ch) {
       switch (ch) {
          case '0' -> { return 0; }
          case '1' -> { return 1; }
          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 'A', 'a' -> { return 10; } // Note: Handle
both upper and lower case.
          case 'B', 'b' -> { return 11; }
          case 'C', 'c' -> { return 12; }
          case 'D', 'd' -> { return 13; }
          case 'E', 'e' -> { return 14; }
          case 'F', 'f' -> { return 15; }
          default  -> { return -1; }
       }
    }  // end hexValue

} // end class Hex2Dec
```

Solution for Programming Exercise 4.3

Exercise 4.3:

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.)

Discussion

The subroutine we have to write is very similar to the program from Exercise 3.1. The main difference is that instead of rolling until both dice come up 1, we roll until the total showing on the dice is equal to some specified value. That value is given by the parameter to the function. Of course, the first thing that the subroutine should do is check that the value of the parameter is in the range of possible rolls of a pair of dice; if not, it should throw an exception. Note that without this check, the subroutine would go into an infinite loop when the parameter is outside the range of possible values. I named the function `rollFor`:

```
public static int rollFor( int N ) {
    if ( N < 2 || N > 12 )
       throw new IllegalArgumentException("Impossible
total for a pair of dice.");
    int die1, die2;  // Numbers between 1 and 6
representing the dice.
    int roll;        // Total showing on dice.
    int rollCt;      // Number of rolls made.
    rollCt = 0;
    do {
       die1 = (int)(Math.random()*6) + 1;
```

```
        die2 = (int)(Math.random()*6) + 1;
        roll = die1 + die2;
        rollCt++;
     } while ( roll != N );
     return rollCt;
}
```

In the actual program, I've added an appropriate Javadoc comment.

You should understand the contract of this subroutine. The parameter, `N`, is supposed to be one of the numbers that could possibly come up on a pair of dice. That is, `N` must be one of 2, 3, ..., or 12. The condition that `N` have a valid value is a precondition for the subroutine. If the caller of the function violates this precondition, the subroutine can't give any sort of correct answer, so it responds by throwing an exception.

The `main()` routine for this program is trivial. In fact, it could even be shortened to:

```
public static void main(String[] args) {
   System.out.println("It took " + rollFor(2) + " rolls to
get snake eyes.");
}  // end main()
```

The Solution

```
/**
 * This program simulates rolling a pair of dice over and
over until the
 * total showing on the two dice is 2.  It reports the
number of rolls
 * it took to get a 2.  (This was written to test the
subroutine, rollFor.)
 */
public class RollFor2 {

   public static void main(String[] args) {
      int numberOfRolls;  // Number of rolls to get a 2.
      numberOfRolls = rollFor(2);
      System.out.println("It took " + numberOfRolls + "
rolls to get snake eyes.");
   }  // end main()
```

```
   /**
    * Simulates rolling a pair of dice until a given total
comes up.
    * Precondition:  The desired total is between 2 and 12,
inclusive.
    * @param N the total that we want to get on the dice
    * @return the number of times the dice are rolled
before the
    *    desired total occurs
    * @throws IllegalArgumentException if the parameter, N,
is not a number
    *    that could possibly come up on a pair of dice
    */
   public static int rollFor( int N ) {
       if ( N < 2 || N > 12 )
          throw new IllegalArgumentException("Impossible
total for a pair of dice.");
       int die1, die2;  // Numbers between 1 and 6
representing the dice.
       int roll;        // Total showing on dice.
       int rollCt;      // Number of rolls made.
       rollCt = 0;
       do {
          die1 = (int)(Math.random()*6) + 1;
          die2 = (int)(Math.random()*6) + 1;
          roll = die1 + die2;
          rollCt++;
       } while ( roll != N );
       return rollCt;
   }

}  // end class RollFor2
```

Solution for Programming Exercise 4.4

Exercise 4.4:

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
       .                  .
       .                  .
```

Discussion

The solution uses the subroutine, `rollFor`, from Exercise 4.3. That subroutine will throw an exception if its parameter is not valid. However, in my program, I **know** that the values that I pass to the `rollFor` subroutine are valid and that no exception will occur. So, there is no need to use a `try..catch` statement to handle the exception.

The `main()` program simply prints a heading for the output, then uses a `for` loop to compute and print the data for each of the possible rolls from 2 to 12. It is not difficult to write it, with a little care to get the formatting right (using formatted output).

The only thing left is to write a function to find the average number of rolls to get a given total on the dice. The average will be a real number, so the return type of the function is double. The subroutine has a parameter of type int that specifies the number we are rolling for. I'll call the parameter "`roll`". An algorithm for the subroutine is

```
Let totalRolls = 0
Repeat 10000 times:
   Call rollFor(roll) to run the experiment once
   Add the returned value to totalRolls
Compute the average by dividing totalRolls by 10000
Return the average
```

In my program, I use a named constant, `NUMBER_OF_EXPERIMENTS`, to specify the number of experiments to be performed. This constant replaces the value 10000, making it easier to read the program and easier to change the number of experiments if I decide I want to do more experiments or fewer. This gives the subroutine:

```
public static double getAverageRollCount( int roll ) {
    int rollCountThisExperiment;  // Number of rolls in
one experiment.
    int rollTotal;  // Total number of rolls in all the
experiments.
    double averageRollCount;  // Average number of rolls
per experiment.
    rollTotal = 0;
    for ( int i = 0;  i < NUMBER_OF_EXPERIMENTS;  i++ ) {
       rollCountThisExperiment = rollFor( roll );
       rollTotal += rollCountThisExperiment;
    }
    averageRollCount = ((double)rollTotal) /
NUMBER_OF_EXPERIMENTS;
    return averageRollCount;
}
```

Note that when the average is computed, a type-cast is used to convert `rollTotal` to type double. This is necessary since `rollCount` and `NUMBER_OF_EXPERIMENTS` are integers, and the computer would evaluate the quotient `rollCount / NUMBER_OF_EXPERIMENTS` as an integer.

By the way, this subroutine could be substantially abbreviated at the expense of being somewhat less easy to understand:

```
public static double getAverageRollCount( int roll ) {
    int rollTotal = 0;  // Total number of rolls in all
the experiments.
    for ( int i = 0;  i < NUMBER_OF_EXPERIMENTS;  i++ )
       rollTotal += rollFor( roll );
    return ((double)rollTotal) / NUMBER_OF_EXPERIMENTS;
}
```

The Solution

```
/**
 * This program performs the following type of experiment:
Given a desired
 * total roll, such as 7, roll a pair of dice until the
given total comes up,
 * and count how many rolls are necessary.  Now do that
over and over, and
 * find the average number of rolls.  The number of times
the experiment is
 * repeated is given by the constant,
NUMBER_OF_EXPERIMENTS.  The average is
 * computed and printed out for each possible roll = 2, 3,
..., 12.
 */

public class DiceRollStats {

   /**
    * The number of times that the experiment "roll for a
given total"
    * is to be repeated.  The program performs this many
experiments, and
    * prints the average of the result, for each possible
roll value,
    */
   public static final int NUMBER_OF_EXPERIMENTS = 10000;

   public static void main(String[] args) {
       double average;  // The average number of rolls to
get a given total.
       System.out.println("Total On Dice     Average Number
of Rolls");
       System.out.println("-------------     --------------
---------");
       for ( int dice = 2;  dice <= 12;  dice++ ) {
          average = getAverageRollCount( dice );
```

```
          System.out.printf("%10d%22.4f\n", dice, average);
             // Use 10 spaces to output dice, and use 22
spaces to output
             // average, with 4 digits after the decimal.
       }
   }

   /**
    * Find the average number of times a pair of dice must
be rolled to get
    * a given total.  The experiment of rolling for the
given total is
    * repeated NUMBER_OF_EXPERIMENTS times and the average
number of rolls
    * over all the experiments is computed.
    * Precondition:  The given total must be between 2 and
12, inclusive.
    * @param roll the total that we want to get on the dice
    * @return the average number of rolls that it takes to
get the specified
    *    total
    */
   public static double getAverageRollCount( int roll ) {
       int rollCountThisExperiment;  // Number of rolls in
one experiment.
       int rollTotal;  // Total number of rolls in all the
experiments.
       double averageRollCount;  // Average number of rolls
per experiment.
       rollTotal = 0;
       for ( int i = 0;  i < NUMBER_OF_EXPERIMENTS;  i++ )
{
          rollCountThisExperiment = rollFor( roll );
          rollTotal += rollCountThisExperiment;
       }
       averageRollCount = ((double)rollTotal) /
NUMBER_OF_EXPERIMENTS;
       return averageRollCount;
   }

   /**
    * Simulates rolling a pair of dice until a given total
comes up.
    * Precondition:  The desired total is between 2 and 12,
inclusive.
    * @param N the total that we want to get on the dice
    * @return the number of times the dice are rolled
before the
```

```
    *    desired total occurs
    * @throws IllegalArgumentException if the parameter, N,
is not a number
    *    that could possibly come up on a pair of dice
    */
   public static int rollFor( int N ) {
       if ( N < 2 || N > 12 )
          throw new IllegalArgumentException("Impossible
total for a pair of dice.");
       int die1, die2;  // Numbers between 1 and 6
representing the dice.
       int roll;        // Total showing on dice.
       int rollCt;      // Number of rolls made.
       rollCt = 0;
       do {
          die1 = (int)(Math.random()*6) + 1;
          die2 = (int)(Math.random()*6) + 1;
          roll = die1 + die2;
          rollCt++;
       } while ( roll != N );
       return rollCt;
   }

}  // end DiceRollStats
```

Solution for Programming Exercise 4.5

Exercise 4.5:

This exercise asks you to write a few lambda expressions and a function that returns a lambda expression as its value. Suppose that a function interface *ArrayProcessor* is defined as

```
public interface ArrayProcessor {
    double apply( double[] array );
}
```

Write a class that defines four `public static final` variables of type *ArrayProcessor* that process an array in the following ways: find the maximum value in the array, find the minimum value in an array, find the sum of the values in the array, and find the average of the values in the array. In each case, the value of the variable should be given by a lambda expression. The class should also define a function

```
public static ArrayProcessor counter( double value ) { ...
```

This function should return an *ArrayProcessor* that counts the number of times that *value* occurs in an array. The return value should be given as a lambda expression.

The class should have a *main*() routine that tests your work. The program that you write for this exercise will need access to the file ArrayProcessor.java, which defines the functional interface.

Discussion

The interface *ArrayProcessor* defines a function `apply(array)` that takes an array of type `double[]` and that returns a double. A lambda expression of type *ArrayList* will typically take the form

```
array -> {
    statements-to-compute-value
    return a-double-value;
}
```

(Of course, the name of the parameter, *array*, could be anything.) For example, a lambda expression for finding the sum of the numbers in the array could be written

```
array -> {
    double total = 0;
    for (int i = 0; i < array.length; i++) {
        total = total + array[i];
    }
    return total;
}
```

This lambda expression is a value of type *ArrayProcessor*. For this exercise, we are supposed to assign this lambda expression to a `public static final` member variable of type *ArrayProcessor*. Calling that variable *sumer*, this takes the form:

```
public static final ArrayProcessor sumer = array -> {
    double total = 0;
    for (int i = 0; i < array.length; i++) {
        total += array[i];
    }
    return total;
};
```

Note the semicolon at the end. The semicolon is not part of the lambda expression; it marks the end of the assignment statement.

Lambda expressions for the minimum and maximum of an array can be handled in a similar way. So could the average, but the average of an array is just the sum of the array divided by its length. Since we already have the *ArrayProcessor* *sumer* for computing the sum of an array, we might as well use it for computing the average. To apply *sumer* to an array, *A*, of type `double[]`, we have to call the *apply*() function that is defined by the *ArrayProcessor* interface: *sumer.apply*(*A*). So, we can define an *ArrayProcessor* for computing the average of an array by:

```
public static final ArrayProcessor averager =
            array -> sumer.apply(array) / array.length;
```

When counting the number of times that a value appears in array, the answer depends on *which* value is being counted. That is, an *ArrayProcessor* that counts the number of times a value appears in an array is a function of the value that is being counted. The *counter* function that we are asked to write can simply return an appropriate lambda expression:

```
public static ArrayProcessor counter( double value ) {
    return array -> {
        int count = 0;
        for (int i = 0; i < array.length; i++) {
            if ( array[i] == value )
                count++;
        }
        return count;
    };
}
```

Note that the parameter, *value*, is used in the definition of the array. So, for example, *counter*(17.0) will return an *ArrayProcessor* whose *apply*() method will count the number of times 17.0 occurs in an array. In my solution, I use that *apply* method in an expression of the form `counter(17.0).apply(secondList)`, where *secondList* is a variable of type `double[]`. Take a look at the *main*() routine in the program below to see how the other *ArrayProcessors* are used.

The Solution

```
/**
 * This class defines several public static member
variables of
 * type ArrayProcessor that process arrays in various ways.
It
 * also defines a function that can create ArrayProcessors
for
 * counting occurrences of values in an array.  (Note that
this
 * program depends on interface ArrayProcessor.)
 */
public class LambdaTest {
```

```
    /**
     * This function returns an ArrayProcessor that counts
     * the number of times a certain value occurs in an
array
     * of doubles.  The parameter specifies the value that
is
     * to be counted.
     */
    public static ArrayProcessor counter( double value ) {
        return array -> {
            int count = 0;
            for (int i = 0; i < array.length; i++) {
                if ( array[i] == value )
                    count++;
            }
            return count;
        };
    }


    /**
     * An ArrayProcessor that computes and returns the
maximum
     * value of an array.  (The array must have length at
least 1.)
     */
    public static final ArrayProcessor maxer = array -> {
        double max = array[0];
        for (int i = 0; i < array.length; i++) {
            if ( array[i] > max)
                max = array[i];
        }
        return max;
    };


    /**
     * An ArrayProcessor that computes and returns the
minimum
     * value of an array.  (The array must have length at
least 1.)
     */
    public static final ArrayProcessor miner = array -> {
        double min = array[0];
        for (int i = 0; i < array.length; i++) {
            if ( array[i] < min)
                min = array[i];
```

```
        }
        return min;
    };


    /**
     * An ArrayProcessor that computes and returns the sum
of the
     * values in an array.  (The array must have length at
least 1.)
     */
    public static final ArrayProcessor sumer = array -> {
        double total = 0;
        for (int i = 0; i < array.length; i++) {
            total += array[i];
        }
        return total;
    };


    /**
     * An ArrayProcessor that computes and returns the
average of the
     * values in an array.  (The array must have length at
least 1.)
     */
    public static final ArrayProcessor averager =
            array -> sumer.apply(array) / array.length;



    //-------------------------------------------------------
---------------------------------------------

    /**
     * A main() routine to test the (other) public members
of this class.
     */
    public static void main(String[] args) {

        double[] firstList = { 1, 2, 3, 4, 5, 6, 7, 8, 9,
10 };
        double[] secondList = { 17.0, 3.14, 17.0, -3.4,
17.0, 42.0, 29.2, 3.14 };

        System.out.println("Sum of first list (should be
55): "
                                                        +
```

```
sumer.apply(firstList) );
        System.out.println("Average of first list (should
be 5.5): "
                                                        +
averager.apply(firstList) );
        System.out.println("Minimum of second list (should
be -3.4): "
                                                        +
miner.apply(secondList) );
        System.out.println("Maximum of second list (should
be 42.0): "
                                                        +
maxer.apply(secondList) );

        System.out.println();

        System.out.println("Count of 17.0 in second list
(should be 3): "
                                                        +
counter(17.0).apply(secondList) );
        System.out.println("Count of 20.0 in second list
(should be 0): "
                                                        +
counter(20.0).apply(secondList) );
        System.out.println("Count of 5.0 in first list
(should be 1): "
                                                        +
counter(5.0).apply(firstList) );

    }


} // end class LambdaTest
```

Solution for Programming Exercise 4.6

Exercise 4.6:

The sample program RandomMosaicWalk.java from Section 4.7 shows a "disturbance" that wanders around a grid of colored squares. When the disturbance visits a square, the color of that square is changed. Here's an idea for a variation on that program. In the new version, 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. The result will be a visually interesting effect, as the path followed by the disturbance gradually turns a brighter and brighter green.

Write a subroutine that will add 25 to the green component of one of the squares in the mosaic. (But don't let the green component go over 255, since that's the largest legal value for a color component.) 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 4.8 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.

By default, the rectangles in the mosaic have a "3D" appearance and a gray border that makes them look nicer in the random walk program. But for this program, you want to turn off that effect. To do so, call `Mosaic.setUse3DEffect(false)` in the main program.

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.

Discussion

This is an exercise in making a rather small modification to a relatively complicated existing program.

The only real problem is to write a new subroutine, which I will call `brightenSquare`. Much of the program comes directly from RandomMosaicWalk2.java. The `randomMove()` routine is unchanged. The important changes in the `main()` routine are to substitute a call to `brightenSquare` for the call to `changeToRandomColor` and to add the line calling `Mosaic.setUse3DEffect`. I also set the delay to 1 in the call to `Mosaic.delay()` to speed up the program. The subroutines `fillWithRandomColors` and `changeToRandomColor` in the `RandomMosaicWalk2` program are not needed in the new program and should be removed. In the three lines that define the constants, the values are changed according the instructions in the exercise:

```
final static int ROWS = 80;         // Number of rows in
the mosaic.
final static int COLUMNS = 80;      // Number of columns in
the mosaic.
final static int SQUARE_SIZE = 5;   // Size of each square
in the mosaic.
```

With these values, the program is interesting to watch for a while. You might want to try using shades of red, blue, or gray, instead of green. Or even use three disturbances, one incrementing the red component of the color, one incrementing the green component, and one incrementing the blue.

An outline for the `brightenSquare` routine is clear, remembering that the green component can't be more than 255:

```
Let g be the current green component of the square
Add 25 to g
If g is greater than 255
    set g = 255
Set the color components of the square to  0, g, 0
```

The red and blue components of the color will always be zero. However, they must be specified in the `Mosaic.setColor()` routine. Written in Java,

the body of the routine is just three lines long:

```
static void brightenSquare(int row, int col) {
    int g = Mosaic.getGreen(row,col);
    g += 25;
    if (g > 255)
        g = 255;
    Mosaic.setColor(row,col,0,g,0);
}
```

The Solution

```
/**
 * This program opens a Mosaic window that is initially
filled with black.
 * A "disturbance" moves randomly around in the window.
Each time it visits
 * a square, the green component of the color of that
square is increased
 * until, after about ten visits, it has reached the
maximum possible level.
 * The animation continues until the user closes the
window.
 */

public class RandomBrighten {

   final static int ROWS = 80;        // Number of rows in
the mosaic.
   final static int COLUMNS = 80;     // Number of columns
in the mosaic.
   final static int SQUARE_SIZE = 5;  // Size of each
square in the mosaic.

   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( ROWS, COLUMNS, SQUARE_SIZE, SQUARE_SIZE
);
      currentRow = ROWS / 2;   // start at center of window
      currentColumn = COLUMNS / 2;
      Mosaic.setUse3DEffect(false);
      while ( true ) {
         brightenSquare(currentRow, currentColumn);
         randomMove();
         Mosaic.delay(1);
      }
   }  // end main

   /**
    * Add a bit of green to the rectangle in a given row
and column.
    * Precondition:   The specified rowNum and colNum are
in the valid range
    *                 of row and column numbers.
    * Postcondition:  The green component of the color of
the square has
    *                 been increased by 25, except that it
does not go
    *                 over its maximum possible value, 255.
    */
   static void brightenSquare(int row, int col) {
      int g = Mosaic.getGreen(row,col);
      g += 25;
      if (g > 255)
          g = 255;
      Mosaic.setColor(row,col,0,g,0);
   }

   /**
    * 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 = ROWS - 1;
          }
          case 1 -> {  // move right
             currentColumn++;
             if (currentColumn >= COLUMNS)
                currentColumn = 0;
          }
          case 2 -> {  // move down
             currentRow ++;
             if (currentRow >= ROWS)
                currentRow = 0;
          }
          case 3 -> {  // move left
             currentColumn--;
             if (currentColumn < 0)
                currentColumn = COLUMNS - 1;
          }
      }
   }  // end randomMove

} // end class RandomBrighten
```

Solution for Programming Exercise 4.7

Exercise 4.7:

For this exercise, you will do something even more interesting with the *Mosaic* class that was discussed in Section 4.7. (Again, don't forget that you will need Mosaic.java and MosaicCanvas.java to compile and run your program.)

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.

Discussion

I will call the program *RandomConvert*, since the basic operation is to convert one square to be the same color as a neighboring square. An outline for the main program is easy:

```
Open a mosaic window
Fill the mosaic with random colors
while the window is open:
   Select one of the rectangles at random
   Convert the color of one of that rectangle's neighbors
   Short delay
```

We have already seen a subroutine for filling the mosaic with random color, in Subsection 4.7.2. I will also write a subroutine to do the second step in the while loop. There is some question about what it means to "select one of the rectangles at random." A rectangle in the mosaic is specified by a row number and a column number. We can select a random rectangle by choosing a row number and a column number at random. Assuming that `ROWS` and `COLUMNS` are constants that give the number of rows and the number of columns, we can do that by saying

```
int randomRow = (int)(ROWS * Math.random());
int randomColumn = (int)(COLUMNS * Math.random());
```

where I have declared each variable and initialized it in one step, as discussed in Subsection 4.8.1. For the "convert" subroutine to do its work, we will have to tell it which rectangle has been selected, so `randomRow` and `randomColumn` will be parameters to that subroutine. The code for the program's `main()` routine becomes:

```
Mosaic.open(ROWS, COLUMNS, SQUARE_SIZE, SQUARE_SIZE);
fillWithRandomColors();
while ( true ) {
   int randomRow = (int)(ROWS * Math.random());
   int randomColumn = (int)(COLUMNS * Math.random());
   convertRandomNeighbor(randomRow, randomColumn);
   Mosaic.delay(1);
}
```

All that remains is to write the `convertRandomNeighbor()` subroutine. This routine should pick a random neighbor of a given rectangle and change its color. A rectangle in the mosaic has four neighbors, above, below, to the left, and to the right. We can pick one at random by selecting a random integer less than four and using that integer to decide which neighbor to select. We have a problem, though, if the rectangle is on the edge of the mosaic. For example, if the rectangle is in the top row, then there is no neighbor **above** that rectangle in the mosaic. One solution to this problem is to wrap around to the bottom of the mosaic and use a square from the bottom row as the neighbor. Essentially, we think of the top of the mosaic as connected to the bottom and the left edge as connected to the right. We have seen something like this in Subsection 4.7.3, in the `randomMove()` subroutine. The `convertRandomNeighbor()` code can use some basic ideas from

`randomMove()`. Here is a version of `convertRandomNeighbor()` that would work:

```
static void convertRandomNeighbor(int row, int col) {

   /* Choose a random direction, and get the row and
column
    * numbers of the neighbor that lies in that direction.
*/

   int neighborRow;       // row number of selected
neighbor
   int neighborColumn;    // column number of selected
neighbor

   int directionNum = (int)(4*Math.random());  // random
direction;

   switch (directionNum) {
       case 0 -> {   // Choose neighbor above.
          neighborColumn = col;        // Neighbor is in
the same column.
          neighborRow = row - 1;       // Subtract 1 to
get neighbor's row number.
          if (neighborRow < 0)         // Neighbor's row
number is outside the mosaic.
             neighborRow = ROWS - 1;  // So wrap around to
bottom of the mosaic.
       }
       case 1 -> {    // Choose neighbor to the right.
          neighborRow = row;               // Same row.
          neighborColumn = col + 1;        // Column to
the right.
          if (neighborColumn >= COLUMNS)   // Outside the
mosaic?
             neighborColumn = 0;          // Wrap around
to the left edge
       }
       case 2 -> {    // Choose neighbor below.
          neighborColumn = col;
          neighborRow = row + 1;
          if (neighborRow >= ROWS)
             neighborRow = 0;
       }
       default -> {    // Choose neighbor to the left.
          neighborRow = row;
          neighborColumn = col - 1;
```

```
            if (neighborColumn < 0)
               neighborColumn = COLUMNS - 1;
         }
      }

      /* Get the color components for position (row,col) */

      int red = Mosaic.getRed(row,col);
      int green = Mosaic.getGreen(row,col);
      int blue = Mosaic.getBlue(row,col);

      /* Change the color of the neighbor to color of the
original square. */


Mosaic.setColor(neighborRow,neighborColumn,red,green,blue)
;

}
```

Note the use of a `default` case at the end of the `switch` statement. Saying "`case 3`" will not work here, because the computer would not be able to verify that values have definitely been assigned to `neighborRow` and `neighborColumn`.

In my program, I actually used a different algorithm that requires somewhat less code. My algorithm goes like this:

```
Get the color components for the rectangle at position
(row,col).
Modify the value of row or col to point to a neighboring
rectangle.
Set the color of position (row,col).
```

This is a little tricky, since the variables `row` and `col` are used both for getting the color and for setting it. But by the time `row` and `col` are used for setting the color, they are referring to a different rectangle. You can see my version of `convertRandomNeighbor` in the full source code listing below.

The Solution

```
/**
 * This program fills a mosaic with random colors.  It then
enters
 * a loop in which it randomly selects one of the squares
in the
 * mosaic, then randomly selects one of the four neighbors
of that
 * square and converts the selected neighbor to the color
of the
 * originally selected square.  The effect is to gradually
build
 * up larger patches of uniform color.  The animation
continues
 * until the user closes the window.  This program depends
on
 * the non-standard classes Mosaic and MosaicCanvas.
 */

public class RandomConvert {

    final static int ROWS = 40;        // Number of rows in
the mosaic.
    final static int COLUMNS = 40;     // Number of columns
in the mosaic.
    final static int SQUARE_SIZE = 10; // Size of each
square in the mosaic.


    /**
     * The main() routine opens the mosaic window, then
enters into
     * a loop in which it repeatedly converts the color of
one square.
     * The loop ends when the user closes the mosaic
window.
     */
    public static void main(String[] args) {
        Mosaic.setUse3DEffect(false);
        Mosaic.open(ROWS, COLUMNS, SQUARE_SIZE,
SQUARE_SIZE);
        fillWithRandomColors();
        while ( true ) {
            int randomRow = (int)(ROWS * Math.random());
            int randomColumn = (int)(COLUMNS *
Math.random());
            convertRandomNeighbor(randomRow, randomColumn);
            Mosaic.delay(1);
        }
```

```
    }


    /**
     * Set each square in the mosaic to be a randomly
selected color.
     */
    static void fillWithRandomColors() {
        for (int row = 0; row < ROWS; row++) {
            for (int col = 0; col < COLUMNS; col++) {
                int r = (int)(256*Math.random());
                int g = (int)(256*Math.random());
                int b = (int)(256*Math.random());
                Mosaic.setColor(row,col,r,g,b);
            }
        }
    }


    /**
     * Select one of the neighbors of the square at
position (row,column) in
     * the mosaic.  Change the color at position (row,
column) to match the
     * color of the selected neighbor.   The neighbors of a
square are the
     * squares above, below, to the left, and to the right
of the square.
     * For squares on the edge of the mosaic, wrap around
to the opposite
     * edge.
     */
    static void convertRandomNeighbor(int row, int col) {

        /* Get the color components for position (row,col)
*/

        int red = Mosaic.getRed(row,col);
        int green = Mosaic.getGreen(row,col);
        int blue = Mosaic.getBlue(row,col);

        /* Choose a random direction, and change the value
of row
         * or col to refer to the neighbor that lies in
that direction. */

        int directionNum = (int)(4*Math.random());
        switch (directionNum) {
```

```
            case 0 -> {    // Choose neighbor above.
                row--;  // Move row number one row up.
                if (row < 0)  // row number is outside the
mosaic.
                    row = ROWS - 1;  // Wrap around to
bottom of the mosaic.
            }
            case 1 -> {    // Choose neighbor to the right.
                col++;
                if (col >= COLUMNS)
                    col = 0;
            }
            case 2 -> {    // Choose neighbor below.
                row++;
                if (row >= ROWS)
                    row = 0;
            }
            case 3 -> {    // Choose neighbor to the left.
                col--;
                if (col < 0)
                    col = COLUMNS - 1;
            }
        }

        /* Change the color of the neighbor to color of the
original square. */

        Mosaic.setColor(row,col,red,green,blue);
    }

} // end class RandomConvert
```

Solution for Programming Exercise 4.8

Exercise 4.8:

Write a program that administers a basic addition quiz to the user. There should be ten questions. Each question is a simple addition problem such as `17 + 42`, where the numbers in the problem are chosen at random (and are not too big). The program should ask the user all ten questions and get the user's answers. After asking all the questions, the user should print each question again, with the user's answer. If the user got the answer right, the program should say so; if not, the program should give the correct answer. At the end, tell the user their score on the quiz, where each correct answer counts for ten points.

The program should use three subroutines, one to create the quiz, one to administer the quiz, and one to grade the quiz. It can use three arrays, with three global variables of type `int[]`, to refer to the arrays. The first array holds the first number from every question, the second holds the second number from every questions, and the third holds the user's answers.

Discussion

The statement of the problem has already made a lot of decisions for us. We are told to use three global variables of type `int[]`. The variables must be `static`, and I have chosen to make them `private`, so the declarations become:

```
private static int[] firstNumbers;  // The first numbers
in all ten questions.
private static int[] secondNumbers; // The second numbers
in all ten questions.
private static int[] userAnswers;   // The user's answers
to the ten questions.
```

Note that these declarations are not in any subroutine. These statements only create variables, not arrays. We still need to create three arrays of length 10. The arrays can be created using the `new` operator. This initialization could in fact be combined with the declaration of the variables:

```
private static int[] firstNumbers = new int[10];
private static int[] secondNumbers = new int[10];
private static int[] userAnswers = new int[10];
```

Another possibility would be to create the three arrays at the beginning of the `main()` routine. In fact, I chose to create them in the subroutines where they are first used.

The program has three stages—create the problems, administer the quiz, and grade the quiz—and we are told to write a subroutine to do each stage. The main program simply has to call the subroutines:

```
public static void main(String[] args) {
    createQuiz();
    administerQuiz();
    gradeQuiz();
}
```

(However, I added a couple of output statements at the start.) This is an example of using subroutines to break up a large task into several simpler subtasks, so that programming each subtask becomes a separate problem. The subroutines are short and not too hard to write. Looking at `administerQuiz()` as an example, note that the user's answers have to be stored in the array, `userAnswers`, so that they can be used when grading the quiz. I create the array itself before asking any questions, and then store the user's responses into the array as they are read from the user. On the other hand, the arrays that hold the numbers that occur in the questions have already been created and filled before `administerQuiz()` is called. Here's the subroutine:

```
private static void administerQuiz() {
    userAnswers = new int[10];
    for (int i = 0; i < 10; i++) {
        int questionNum = i + 1;
        System.out.printf("Question %2d:  What is %2d +
%2d  ? ",
```

```
                        questionNum, firstNumbers[i],
secondNumbers[i]);
        userAnswers[i] = TextIO.getlnInt();
    }
}
```

A couple notes about the output: I wanted to number the questions 1, 2, ..., 10, but the elements of the arrays are numbered 0, 1, ..., 9. This means that the question number that I output has to be one more than the array index. Also, all the numbers that occur in my program are either one or two digits. By allowing exactly two digits for each number in the output, I can get everything to line up neatly. Here's the I/O for a sample run of the program:

```
Welcome to the addition quiz!

Question  1:  What is 40 + 14  ? 54
Question  2:  What is 18 + 38  ? 46
Question  3:  What is 21 + 31  ? 52
Question  4:  What is 18 + 36  ? 54
Question  5:  What is 43 + 12  ? 55
Question  6:  What is 21 + 17  ? 38
Question  7:  What is 33 + 25  ? 58
Question  8:  What is 39 + 21  ? 70
Question  9:  What is  1 + 25  ? 26
Question 10:  What is 39 + 26  ? 65

Here are the correct answers:
   Question  1:  40 + 14  =  54.  You were CORRECT.
   Question  2:  18 + 38  =  56.  You said 46, which is
INCORRECT.
   Question  3:  21 + 31  =  52.  You were CORRECT.
   Question  4:  18 + 36  =  54.  You were CORRECT.
   Question  5:  43 + 12  =  55.  You were CORRECT.
   Question  6:  21 + 17  =  38.  You were CORRECT.
   Question  7:  33 + 25  =  58.  You were CORRECT.
   Question  8:  39 + 21  =  60.  You said 70, which is
INCORRECT.
   Question  9:   1 + 25  =  26.  You were CORRECT.
   Question 10:  39 + 26  =  65.  You were CORRECT.

You got 8 questions correct.
Your grade on the quiz is 80
```

I noted in Subsection 4.3.8 that it's considered bad style to use global variables where parameters could be used instead, and that some people are quite hostile to unnecessary global variables. You might ask, how bad was it to use global variables in this case? To some extent this is a matter of opinion. You could argue that the whole program revolves around the three arrays, so it makes sense for them to be global in the program. On the other hand, it is possible to avoid using globals in this case. The three array variables can be declared in `main()` and the three arrays can be created there and passed as parameters to the subroutines. Here is a main routine that does that:

```
public static void main(String[] args) {
    int[] firstNums = new int[10];  // The first numbers
in the ten problems
    int[] secondNums = new int[10]; // The second numbers
in the ten problems
    int[] answers = new int[10];    // The user's answers.
    System.out.println();
    System.out.println("Welcome to the addition quiz!");
    System.out.println();
    createQuiz(firstNums,secondNums);
    administerQuiz(firstNums,secondNums,answers);
    gradeQuiz(firstNums,secondNums,answers);
}
```

A complete second version of the program, with no global variables, is given below. One reason that I asked you to use global variables is that understanding exactly how arrays work as parameters requires an understanding of objects and how they are passed as parameters. Without that understanding, it's hard to get the global-less version correct. You will learn about passing objects as parameters in the next chapter.

The Solution

Here is the original program, using global variables:

```
import textio.TextIO;

/**
 * This program administers a ten-question addition quiz to
```

```
the user.  The numbers
 * for the problem are chosen at random.  The numbers and
the answers are one or
 * two digits.  After asking the user the ten questions,
the computer grades the
 * quiz, telling the user the correct answer for any
problem they got wrong.
 */
public class AdditionQuiz {

    private static int[] firstNumbers;  // The first
numbers in all ten questions.
    private static int[] secondNumbers; // The second
numbers in all ten questions.
    private static int[] userAnswers;   // The user's
answers to the ten questions.


    public static void main(String[] args) {
        System.out.println();
        System.out.println("Welcome to the addition
quiz!");
        System.out.println();
        createQuiz();
        administerQuiz();
        gradeQuiz();
    }


    /**
     * Creates the arrays that hold the numbers for the
questions and fills
     * them with random numbers.
     */
    private static void createQuiz() {
        firstNumbers = new int[10];
        secondNumbers = new int[10];
        for ( int i = 0; i < 10; i++ ) {
            firstNumbers[i] = (int)(Math.random() * 50 +
1);  // in the range 1 to 50
            secondNumbers[i] = (int)(Math.random() * 50);
// in the range 0 to 49
        }
    }


    /**
     * Asks the user each of the ten quiz questions and
```

```
gets the user's answers.
     * The answers are stored in an array, which is created
in this subroutine.
     */
    private static void administerQuiz() {
        userAnswers = new int[10];
        for (int i = 0; i < 10; i++) {
            int questionNum = i + 1;
            System.out.printf("Question %2d:  What is %2d +
%2d  ? ",
                            questionNum, firstNumbers[i],
secondNumbers[i]);
            userAnswers[i] = TextIO.getlnInt();
        }
    }


    /**
     * Shows all the questions, with their correct answers,
and computes a grade
     * for the quiz.  For each question, the user is told
whether they got
     * it right.
     */
    private static void gradeQuiz() {
        System.out.println();
        System.out.println("Here are the correct
answers:");
        int numberCorrect = 0;
        int grade;
        for (int i = 0; i < 10; i++) {
            int questionNum = i + 1;
            int correctAnswer = firstNumbers[i] +
secondNumbers[i];
            System.out.printf("   Question %2d:  %2d + %2d
=  %2d.  ",
                questionNum, firstNumbers[i],
secondNumbers[i], correctAnswer);
            if ( userAnswers[i] == correctAnswer ) {
                System.out.println("You were CORRECT.");
                numberCorrect++;
            }
            else {
                System.out.println("You said " +
userAnswers[i] + ", which is INCORRECT.");
            }
        }
        grade = numberCorrect * 10;
```

```
        System.out.println();
        System.out.println("You got " + numberCorrect + "
questions correct.");
        System.out.println("Your grade on the quiz is " +
grade);
        System.out.println();
    }

} // end class AdditionQuiz
```

Here is a version that uses parameters and no global variables:

```
import textio.TextIO;

/**
 * This program administers a ten-question addition quiz to
the user.  The numbers
 * for the problem are chosen at random.  The numbers and
the answers are one or
 * two digits.  After asking the user the ten questions,
the computer grades the
 * quiz, telling the user the correct answer for any
problem they got wrong.
 */
public class AdditionQuizNoGlobals {


    public static void main(String[] args) {
        int[] firstNums = new int[10];  // The first
numbers in the ten problems
        int[] secondNums = new int[10]; // The second
numbers in the ten problems
        int[] answers = new int[10];    // The user's
answers.
        System.out.println();
        System.out.println("Welcome to the addition
quiz!");
        System.out.println();
        createQuiz(firstNums,secondNums);
        administerQuiz(firstNums,secondNums,answers);
        gradeQuiz(firstNums,secondNums,answers);
    }
```

```
    /**
     * Creates the arrays that hold the numbers for the
questions and fills
     * them with random numbers.  The parameters are arrays
that will hold
     * the random numbers for the first and second operands
of each addition
     * problem.  The arrays must have already been created
when this subroutine
     * is called!
     */
    private static void createQuiz(int[] firstNumbers,
int[] secondNumbers) {
        for ( int i = 0; i < 10; i++ ) {
            firstNumbers[i] = (int)(Math.random() * 50 +
1);  // in the range 1 to 50
            secondNumbers[i] = (int)(Math.random() * 50);
// in the range 0 to 49
        }
    }


    /**
     * Asks the user each of the ten quiz questions and
gets the user's answers.
     * The first two parameters hold the operands for the
quiz questions.  The user's
     * answers to the ten problems will be stored in the
third array.  All arrays
     * exist before this subroutine is called, and the
first two arrays have
     * already been filled with values.
     */
    private static void administerQuiz(int[] firstNumbers,
int[] secondNumbers, int[] userAnswers) {
        for (int i = 0; i < 10; i++) {
            int questionNum = i + 1;
            System.out.printf("Question %2d:  What is %2d +
%2d  ? ",
                            questionNum, firstNumbers[i],
secondNumbers[i]);
            userAnswers[i] = TextIO.getlnInt();
        }
    }


    /**
     * Shows all the questions, with their correct answers,
```

```
and computes a grade
     * for the quiz.  For each question, the user is told
whether they got
     * it right.  The first two parameters hold the
operands for the quiz questions,
     * and the third parameter holds the answers that the
user gave to the quiz.
     */
    private static void gradeQuiz(int[] firstNumbers, int[]
secondNumbers, int[] userAnswers) {
        System.out.println();
        System.out.println("Here are the correct
answers:");
        int numberCorrect = 0;
        int grade;
        for (int i = 0; i < 10; i++) {
            int questionNum = i + 1;
            int correctAnswer = firstNumbers[i] +
secondNumbers[i];
            System.out.printf("   Question %2d:  %2d + %2d
=  %2d.  ",
                questionNum, firstNumbers[i],
secondNumbers[i], correctAnswer);
            if ( userAnswers[i] == correctAnswer ) {
                System.out.println("You were CORRECT.");
                numberCorrect++;
            }
            else {
                System.out.println("You said " +
userAnswers[i] + ", which is INCORRECT.");
            }
        }
        grade = numberCorrect * 10;
        System.out.println();
        System.out.println("You got " + numberCorrect + "
questions correct.");
        System.out.println("Your grade on the quiz is " +
grade);
        System.out.println();
    }

}  // end class AdditionQuizNoGlobals
```

Solution for Programming Exercise 5.1

Exercise 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, including at least a `toString()` method. 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.

Discussion

The versions of the *PairOfDice* class in Section 2 differ in how the dice are initialized. I like the idea of initializing the dice to random values, so I will work with the following version:

```
public class PairOfDice {

    public int die1;   // Number showing on the first die.
    public int die2;   // Number showing on the second
die.

    /**
     * Constructor creates a pair of dice and rolls them
so that
     * they initially show some random value.
     */
    public PairOfDice() {
        roll();  // Call the roll() method to roll the
dice.
    }
```

```
    /**
     * Roll the dice by setting each die to be a random
number between 1 and 6.
     */
    public void roll() {
        die1 = (int)(Math.random()*6) + 1;
        die2 = (int)(Math.random()*6) + 1;
    }

} // end class PairOfDice
```

After a *PairOfDice* object has just been created, the number on each die is definitely between 1 and 6, just like the number on a real die. Can we be sure that this will always be true? Not if the instance variables `die1` and `die2` are `public`, since they can be changed from outside the class. There is nothing to stop someone from changing them to 42 and -17 or anything else. It's not good enough to say that you're not supposed to do that with dice. I want an absolute guarantee that my dice objects can only have the values that real dice could have. By making `die1` and `die2` `private`, I can have that guarantee, because the code that I write in the *PairOfDice* class is the only code that will ever directly affect the values of the variables.

So, we will make `die1` and `die2` private, and add instance methods `getDie1()` and `getDie2()` to return the values of `die1` and `die2`. These are "getter" methods that get the value of a member variable, and their names follow the convention that the name of a getter method consists of "get" followed by a capitalized version of the variable name. It would still be useful to be able to set the values of the dice, but we want to make it impossible to set the value to anything outside the range 1 to 6. The solution is to provide "setter" methods that throw an exception when the value is illegal.

I also added the `toString()` method from Subsection 5.3.3. As for other improvements, I can foresee that people who use my class will often be interested in the total on the dice, so they will tend to say things like "`dice.getDie1() + dice.getDie2()`" a lot. If this is going to be done over and over, why not provide a method in the class to do it? So, I added a method `getTotal()` that returns the total value showing on the two dice. The complete, modified *PairOfDice* class is shown below.

(Here is another improvement that I thought about. We could modify the `roll()` method so that in addition to rolling the dice, it would also return an int value giving the total on the dice. For example, this would allow us to replace "`dice.roll(); val = dice.getTotal()`" with "`val = dice.roll()`". Since it's legal to call a function with a subroutine call statement, we could still say "`dice.roll()`" if we just want to roll the dice without recording the total immediately. However, I decided that it was a little clearer to leave this feature out.)

The main program is easy, especially since we've done the same problem before without using objects (in Exercise 3.1). Note how the pair of dice object is used. To test whether or not the total on the dice is 2, I use the test "`while (dice.getTotal() != 2)`". Because of the `toString()` method, I just have to say say

```
System.out.println("The dice come up " + dice );
```

to show the numbers on the two dice.

The Solution

The Modified PairOfDice Class

```
     /**
      * An object of class PairOfDice represents a pair of
dice,
      * where each die shows a number between 1 and 6.  The
dice
      * can be rolled, which randomizes the numbers showing
on the
      * dice.
      */
     public class PairOfDice {

        private int die1;   // Number showing on the first
die.
        private int die2;   // Number showing on the second
die.
```

```
        /**
         * Constructor creates a pair of dice and rolls
them so that
         * they initially show some random value.
         */
        public PairOfDice() {
            roll();  // Call the roll() method to roll the
dice.
        }

        /**
         * Roll the dice by setting each die to be a random
number between 1 and 6.
         */
        public void roll() {
            die1 = (int)(Math.random()*6) + 1;
            die2 = (int)(Math.random()*6) + 1;
        }

        /**
         * Return the number showing on the first die.
         */
        public int getDie1() {
           return die1;
        }

        /**
         * Set the value of the first die.  Throws an
IllegalArgumentException
         * if the value is not in the range 1 to 6.
         */
        public void setDie1( int value ) {
           if ( value < 1 || value > 6 )
              throw new IllegalArgumentException("Illegal
dice value " + value);
           die1 = value;
        }

        /**
         * Return the number showing on the second die.
         */
        public int getDie2() {
           return die2;
        }

        /**
         * Set the value of the second die.  Throws an
```

```
IllegalArgumentException
         * if the value is not in the range 1 to 6.
         */
        public void setDie2( int value ) {
           if ( value < 1 || value > 6 )
              throw new IllegalArgumentException("Illegal
dice value " + value);
           die2 = value;
        }

        /**
         * Return the total showing on the two dice.
         */
        public int getTotal() {
           return die1 + die2;
        }

        /**
         * 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;
        }

     }  // end class PairOfDice
```

The Main Program

```
     /**
      * Rolls a pair of dice until the dice come up snake
eyes
      * (with a total value of 2).  Counts and reports the
      * number of rolls.
      */
     public class RollFor2 {
```

```
        public static void main(String[] args) {

           PairOfDice dice;  // A variable that will refer
to the dice.
           int rollCount;    // Number of times the dice
have been rolled.

           dice = new PairOfDice();  // Create the
PairOfDice object.
           rollCount = 0;

           /* Roll the dice until they come up snake eyes.
*/

           do {
               dice.roll();
               System.out.println("The dice come up " +
dice );
               rollCount++;
           } while (dice.getTotal() != 2);

           /* Report the number of rolls. */

           System.out.println("\nIt took " + rollCount + "
rolls to get a 2.");

           /* Now, generate an exception. */

           System.out.println();
           System.out.println("This program will now crash
with an error");
           System.out.println("when it tries to set the
value of a die to 42.");
           System.out.println();

           dice.setDie1(42);
           System.out.println(dice);  // This statement
will not be executed!

        }

     }  // end class RollFor2
```

Solution for Programming Exercise 5.2

Exercise 5.2:

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 instance methods are available:

- `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`.

Discussion

For the *StatCalc* class to handle minimums and maximums, some of what must be added to the class is obvious. We need two new instance variables, `min` and `max`, and two getter methods to return the values of those instance variables. So, we can add these lines to the class definition:

```
private double min;   // Smallest item that has been
entered.
private double max;   // Largest item that has been
entered.

public double getMin() {
   return min;
}

public double getMax() {
   return max;
}
```

But then there is the problem of making sure that `min` and `max` have the right values. `min` records the smallest number seen so far. Every time we have a new number to add to the dataset there is a possibility that `min` will change, so we have to compare `min` with the newly added number. If the new number

is smaller than the current `min`, then the number becomes the new value of `min` (since the new number is now the smallest number we have seen so far). We do something similar for `max`. This has to be done whenever a number is entered into the dataset, so it has to be added to the `enter()` method, giving:

```
public void enter(double num) {
      // This is not completely correct!
   count++;
   sum += num;
   squareSum += num*num;
   if (num > max)           // We have a new maximum.
      max = num;
   if (num < min)           // We have a new minimum.
      min = num;
}
```

Unfortunately, if this is all we do, there is a **bug in our program**! For example, if the dataset consists of the numbers 21, 17, and 4, the computer will insist that the minimum is 0, rather than 4. The problem is that the variables `min` and `max` are initialized to zero. (If no initial value is provided for a numerical instance variable, it gets the default initial value, zero.) Since `min` is 0, none of the numbers in the dataset pass the test "`if (num < min)`", and therefore the value of `min` never changes. A similar problem holds for `max`, but it will only show up if all the numbers in the dataset are less than zero. For the other instance variables, `count`, `sum`, and `squareSum`, the default initial value of zero is correct. For `min` and `max`, we have to do something different.

One possible way to fix the problem is to treat the first number entered as a special case. When only one number has been entered, it's certainly the largest number so far and also the smallest number so far, so it should be assigned to both `min` and `max`. This can be handled in the `enter()` method:

```
public void enter(double num) {
      // (This is NOT the version I used in my final
answer.)
   count++;
   sum += num;
   squareSum += num*num;
   if (count == 1) {  // This is the first number.
      max = num;
      min = num;
```

```
    }
    else {
       if (num > max)      // We have a new maximum.
          max = num;
       if (num < min)      // We have a new minimum.
          min = num;
    }
}
```

This works fine. However, I decided to use an alternative approach. We would be OK if we could initialize `min` to have a value that is bigger than any possible number. Then, when the first number is entered, it definitely will satisfy the test "`if (num < min)`", and therefore the first number will become the value of `min`. But to be "bigger than any possible number," `min` would have to be infinity. The initial value for `max` has to be smaller than any possible number, so `max` has to be initialized to negative infinity. And that's what we'll do!

Recall from Section 2.5 that the standard class *Double* contains constants `Double.POSITIVE_INFINITY` and `Double.NEGATIVE_INFINITY` that represent positive and negative infinity. We can use these named constants to provide initial values for the instance variables `min` and `max`. So, the declarations become:

```
private double max = Double.NEGATIVE_INFINITY;  // Largest
item seen.
private double min = Double.POSITIVE_INFINITY;  //
Smallest item seen.
```

With this change, the *StatCalc* class works correctly. The complete class is shown below. (By the way, you might think about what happens if `getMin()` or `getMax()` is called before any data has been entered. What actually happens? What should happen? What is the minimum or maximum of a set of numbers that contains no numbers at all?)

The main program is fairly straightforward, so just for fun I decided to use a *Scanner* instead of *TextIO* to read the user's input (see Subsection 2.4.6. The user's data are read and entered into the *StatCalc* object in a loop:

```
do {
   System.out.print("? ");
```

```
    item = in.nextDouble();  // (in is the Scanner.)
    if (item != 0)
       calc.enter(item);
} while ( item != 0 );
```

The subroutine call "`calc.enter(item);`" enters the user's item. That is, it does all the processing necessary to include this data item in the statistics it is computing. After all the data have been entered, the statistics can be obtained by using function calls such as "`calc.getMean()`". The statistics are output in statements such as:

```
System.out.println("   Average:            " +
calc.getMean());
```

Note that a function call represents a value, and so can be used anyplace where a variable or literal value could be used. I don't have to assign the value of the function to a variable. I can use the function call directly in the output statement. Another note: In this program, I decided not to use formatted output, since it seems appropriate to print the answers with as much accuracy as possible. For formatted output, the statement used to print the mean could be something like:

```
System.out.printf("    Average:            %10.3f%n",
calc.getMean());
```

The complete main program is shown below.

Although that completes the exercise, one might wonder: Instead of modifying the source code of *StatCalc*, could we make a subclass of *StatCalc* and put the modifications in that? The answer is yes. We will need need to use the slightly obscure special variable `super` that was discussed in Subsection 5.6.2.

The new instance variables and instance methods can simply be put into the subclass. The problem arises with the `enter()` method. We have to redefine this method so that it will update the values of `min` and `max`. But it also has to do all the processing that is done by the original `enter()` method in the *StatCalc* class. This is what `super` is for. It lets us call a method from the superclass of the class we are writing. So, the subclass can be written:

```
class StatCalcWithMinMax extends StatCalc {

   private double max = Double.NEGATIVE_INFINITY;  //
Largest item seen.
   private double min = Double.POSITIVE_INFINITY;  //
Smallest item seen.

   public void enter(double num) {
      super.enter(num);  // Call the enter method from the
StatCalc class.
      if (num > max)  // Then do the extra processing for
min and max.
         max = num;
      if (num < min)
         min = num;
   }

   public double getMin() {
      return min;
   }

   public double getMax() {
      return max;
   }

}  // end class StatCalcWithMinMax
```

The Solution

Revised StatCalc Class. Changes from the original are shown in red italic.

```
     /*
      * An object of class StatCalc can be used to compute
several simple statistics
      * for a set of numbers.  Numbers are entered into the
dataset using
      * the enter(double) method.  Methods are provided to
return the following
      * statistics for the set of numbers that have been
entered: The number
      * of items, the sum of the items, the average, the
standard deviation,
      * the maximum, and the minimum.
      */
     public class StatCalc {
```

```
        private int count;   // Number of numbers that have
been entered.
        private double sum;  // The sum of all the items
that have been entered.
        private double squareSum;  // The sum of the
squares of all the items.
        private double max = Double.NEGATIVE_INFINITY;  //
Largest item seen.
        private double min = Double.POSITIVE_INFINITY;  //
Smallest item seen.

        /**
         * Add a number to the dataset.  The statistics
will be computed for all
         * the numbers that have been added to the dataset
using this method.
         */
        public void enter(double num) {
           count++;
           sum += num;
           squareSum += num*num;
           if (num > max)
              max = num;
           if (num < min)
              min = num;
        }

        /**
         * Return the number of items that have been
entered into the dataset.
         */
        public int getCount() {
           return count;
        }

        /**
         * Return the sum of all the numbers that have been
entered.
         */
        public double getSum() {
           return sum;
        }

        /**
         * Return the average of all the items that have
been entered.
```

```
         * The return value is Double.NaN if no numbers
have been entered.
         */
        public double getMean() {
           return sum / count;
        }

        /**
         * Return the standard deviation of all the items
that have been entered.
         * The return value is Double.NaN if no numbers
have been entered.
         */
        public double getStandardDeviation() {
           double mean = getMean();
           return Math.sqrt( squareSum/count - mean*mean );
        }

        /**
         * Return the smallest item that has been entered.
         * The return value will be infinity if no numbers
have been entered.
         */
        public double getMin() {
           return min;
        }

        /**
         * Return the largest item that has been entered.
         * The return value will be -infinity if no numbers
have been entered.
         */
        public double getMax() {
           return max;
        }

     }  // end class StatCalc
```

Main Program

```
     /**
      * Computes and display several statistics for a set
of non-zero
```

```
      * numbers entered by the user.  (Input ends when user
enters 0.)
      * This program uses StatCalc.java.
      */

     import java.util.Scanner;

     public class SimpleStats {

       public static void main(String[] args) {

          Scanner in = new Scanner(System.in);

          StatCalc calc; // Computes stats for numbers
entered by user.
          calc = new StatCalc();

          double item; // One number entered by the user.

          System.out.println("Enter your numbers.  Enter 0
to end.");
          System.out.println();

          do {
             System.out.print("? ");
             item = in.nextDouble();
             if (item != 0)
                calc.enter(item);
          } while (item != 0);

          System.out.println("\nStatistics about your
data:\n");
          System.out.println("   Count:              " +
calc.getCount());
          System.out.println("   Sum:                " +
calc.getSum());
          System.out.println("   Minimum:            " +
calc.getMin());
          System.out.println("   Maximum:            " +
calc.getMax());
          System.out.println("   Average:            " +
calc.getMean());
          System.out.println("   Standard Deviation: "
                 + calc.getStandardDeviation());

       } // end main()

    } // end SimpleStats
```

Solution for Programming Exercise 5.3

Exercise 5.3:

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 required.)

Discussion

The program from Exercise 4.4 defines a function, `rollFor(N)`, that performs the basic experiment once. It rolls the dice until the total on the dice is `N`, and it returns the number of rolls. Using a *PairOfDice* object, `dice`, the body of this subroutine becomes

```
int rollCt = 0;  // Number of rolls made.
do {
   dice.roll();
   rollCt++;
} while ( dice.getTotal() != N );
return rollCt;
```

This is significantly simpler than the original version. But where does the dice object come from? One possibility is to create a new *PairOfDice* object

at the beginning of the function. This will work, but then a new object is created each time the function is called. In the program we are writing, the function is called 110,000 times. It seems a waste to manufacture 110,000 pairs of dice when one would do! To avoid this, I create the dice as a static member variable:

```
private static PairOfDice dice = new PairOfDice();
```

The variable must be `static` since it is used in the `static` function, `rollFor`. Since `dice` is a static member variable, it is created and initialized when the class is first loaded and it exists as long as the program is running. The `rollFor()` method always uses this one pair of dice. (Some people might prefer to create the dice as a local variable in the `main()` routine. The dice could be passed as a parameter to the `rollFor()` method. Then, `rollFor(N,dice)` would mean "roll for a total of `N` using this pair of dice." But in my opinion, the dice are not logically a part of the interface of this subroutine, and I prefer to use the member variable.)

The original program also had a method called `getAverageRollCount()` to find the average number of rolls, when the basic experiment is repeated 10000 times. We could rename this to `getRollCountStats` and use it to compute all the statistics, not just the average. The actual computation is to be done by a *StatCalc* object. Let `stats` be a variable that refers to that object. The results of each experiment will be fed into this object, something like this:

```
for ( int i = 0;  i < 10000;  i++ ) {
      // Assume "total" is the number we are rolling for.
    rollCountThisExperiment = rollFor( total );  // Do one
experiment.
    stats.enter( rollCountThisExperiment );      // Enter
the data.
}
```

At the end of this process, `stats` is ready to report the statistics. All you have to do is call its functions, such as `stats.getMean()`.

In my program, I use the named constant `NUMBER_OF_EXPERIMENTS` instead of the literal number, 10000. I abbreviate the for loop to

```
for ( int i = 0;  i < NUMBER_OF_EXPERIMENTS;  i++ )
    stats.enter( rollFor(total) );
```

and, since it has become so short, I deleted the subroutine and moved the `for` loop into the `main()` routine. The `main()` routine prints the output in neat columns, using formatted output. For example, "`System.out.printf("%6d",total);`" prints the value of `total` in a column of width 6. (The hardest part was figuring out how wide to make the columns!). After printing out some headings for the columns, the `main()` routine says

```
for ( int total = 2;  total <= 12;  total++ ) {
   StatCalc stats;  // An object that will compute the
statistics.
   stats = new StatCalc();
   for ( int i = 0; i < NUMBER_OF_EXPERIMENTS; i++ ) {
         // Do the experiment of counting the number of
rolls
         // required to roll the desired total, and enter
the
         // number of rolls into stats' dataset.
      stats.enter( rollFor(total) );
   }
   System.out.printf("%6d", total);
   System.out.printf("%18.3f", stats.getMean());
   System.out.printf("%19.3f",
stats.getStandardDeviation());
   System.out.printf("%14.3f", stats.getMax());
   System.out.println();
}
```

The body of the `for` loop processes one of the possible totals on a pair of dice, and it produces one line of output with the statistics for that total. A new *StatCalc* object is created for each execution of the `for` loop. This is necessary because we want separate statistics for each total. A single *StatCalc* object would just accumulate all the data for all the possible values of total into a single dataset. It might be worth noting that the five output statements in this method could be written as a single statement:

```
System.out.printf( "%6d%18.3f%19.3f%14.3f%n", total,
stats.getMean(),
        stats.getStandardDeveialtion(), stats.getMax() );
```

By the way, you might wonder what would happen if I had not eliminated the `getRollCountStats()` subroutine? In that case, the statistics data is generated in the subroutine and is used in the `main()` routine. So, the same *StatCalc* object has to be used in both routines. There are several ways to handle this. The variable, `stats`, could be a static member variable. Then it could simply be used in both routines. Alternatively, the *StatCalc* object could be passed as a parameter to the subroutine. Or, as a final and best alternative, the object could be created in the subroutine and sent back to the `main()` routine as a return value. Let's look at this last possibility. The subroutine would be:

```
static StatCalc getRollCountStats( int total ) {
    StatCalc calc;          // An object to compute the
statistics.
    calc = new StatCalc();
    for ( int i = 0; i < NUMBER_OF_EXPERIMENTS; i++ )
        calc.enter( rollFor(total) );
    return calc;  // Send back the object, with the
statistics.
}
```

In the main program, this would be used as follows:

```
for ( int total = 2; total <= 12; total++ ) {
   StatCalc stats;  // The stats for this total
   stats = getRollCountStats( total ); // Get stats from
subroutine.
   System.out.printf("%6d", total);
   System.out.printf("%18.3f", stats.getMean());
   System.out.printf("%19.3f",
stats.getStandardDeviation());
   System.out.printf("%14.3f", stats.getMax());
   System.out.println();
}
```

Note in particular that not every object variable needs to be initialized with a new object. In the case of the variable `calc` in the `main()` routine, an object is computed elsewhere. The variable, `calc`, is set to refer to that existing object.

The Solution

```
/**
 * This program performs the following type of experiment:
 * Given a desired total roll, such as 7, roll a pair of
 * dice until the given total comes up, and count how many
 * rolls are necessary.  Now do the experiment over and
over,
 * and find the average number of rolls.  The number of
times
 * the experiment is repeated is given by the constant,
 * NUMBER_OF_EXPERIMENTS.  Several statistics are computed
and
 * printed out for each possible roll = 2, 3, ..., 12:
 * the average number of rolls, the standard deviation,
 * and the maximum number of rolls.
 */

public class DiceRollStats2 {

   static final int NUMBER_OF_EXPERIMENTS = 10000;

   private static PairOfDice dice = new PairOfDice();
            // A single pair of dice, which will be used
for all
            // the experiments.


   public static void main(String[] args) {

       System.out.println("Dice Total   Avg # of Rolls
Stand. Deviation   Max # of Rolls");
       System.out.println("----------   --------------   --
--------------   --------------");

       for ( int total = 2;  total <= 12;  total++ ) {
          StatCalc stats;  // An object that will compute
the statistics.
          stats = new StatCalc();
          for ( int i = 0; i < NUMBER_OF_EXPERIMENTS; i++ )
{
                // Do the experiment of counting the number
of rolls
                // required to roll the desired total, and
enter the
                // number of rolls into stats' dataset.
             stats.enter( rollFor(total) );
          }
          System.out.printf("%6d", total);
          System.out.printf("%18.3f", stats.getMean());
```

```
          System.out.printf("%19.3f",
stats.getStandardDeviation());
          System.out.printf("%14.3f", stats.getMax());
          System.out.println();
       }

   } // end main

   /**
    * Roll the dice repeatedly until the total on the
    * two dice comes up to be N.  N MUST be one of the
numbers
    * 2, 3, ..., 12.  (If not, this routine will go into an
    * infinite loop!).  The number of rolls is returned.
    */
   static int rollFor( int N ) {
       int rollCt = 0;  // Number of rolls made.
       do {
          dice.roll();
          rollCt++;
       } while ( dice.getTotal() != N );
       return rollCt;
   }


}  // end class DiceRollStats2
```

Solution for Programming Exercise 5.4

Exercise 5.4:

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()`, which 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.

Discussion

This problem is mostly a warm-up for the next one. It uses objects of three different types, *Card*, *Deck*, and *BlackjackHand*. The *Hand* class is used indirectly, as the superclass of *BlackjackHand*. To use these objects, you need to know what methods are available in each class, so you should review the information that you have about the classes before beginning the program.

An algorithm for the program is

```
Create a deck
repeat while user wants to continue:
```

```
    Shuffle the deck
    Create a new BlackjackHand
    Decide the number of cards in the hand
    Deal cards from the deck into the hand, and print them
out
    Display the value of the hand
```

Some variation is possible. You could use just one *BlackjackHand* object, and remove all the cards from it between hands. The *Hand* class includes an instance method, `clear()`, that could be used for this purpose. Similarly, you could create a new *Deck* object each time through the loop. Or, you might want to use one deck and shuffle it only when the number of cards in the deck gets too small. You could say:

```
if (deck.cardsLeft() < 6)
      deck.shuffle();
```

Since we always want to do at least one hand, we can use a `do..while` statement for the loop. Putting in some variable names, we can refine the algorithm to

```
deck  =  new Deck();
do:
    deck.shuffle();
    hand  =  new BlackjackHand();
    cardsInHand = a random number between 2 and 6
    Deal cards from deck into hand, and print them out.
    Display hand.getBlackjackValue()
    Ask if user wants to go again
while user wants to go again
```

The number of cards in the hand is supposed to be a random number between 2 and 6. There are five possible values. The expression "`(int)(Math.random()*5)`" has one of the 5 possible values 0, 1, 2, 3, or 4. Adding 2 to the result gives one of the values 2, 3, 4, 5, or 6. So, the number of cards can be computed as "`2 + (int)(Math.random()*5)`".

Once we know the number of cards, we can use a `for` loop to deal cards into the hand, one at a time. The function call `deck.dealCard()` gets a card from the deck. Once we have a card, we can add it to the hand with the subroutine call `hand.addCard(card)`. This allows us to refine the algorithm to

```
deck  =  new Deck();
do:
    deck.shuffle();
    hand  =  new BlackjackHand();
    cardsInHand = 2 + (int)(Math.random()*5)
    for i = 0 to cardsInHand:
        card  =  deck.dealCard()
        hand.addCard(card)
        Display the card
    Display hand.getBlackjackValue()
    Ask if user wants to go again
while user wants to go again
```

Alternatively, dealing the cards and displaying them could be done in separate `for` loops.

This algorithm can be translated pretty directly into the `main()` routine of the program, which is shown below.

The Solution

```
import textio.TextIO;

/**
 * Creates random blackjack hands, with 2 to 6 cards,
 * and prints out the blackjack value of each hand.
 * The user decides when to stop.
 */
public class TestBlackjackHand {

   public static void main(String[] args) {

      Deck deck;             // A deck of cards.
      Card card;             // A card dealt from the deck.
      BlackjackHand hand;    // A hand of from two to six
cards.
      int cardsInHand;       // Number or cards in the hand.
      boolean again;         // Set to true if user wants to
continue.

      deck = new Deck();     // Create the deck.

      do {
         deck.shuffle();
```

```
         hand = new BlackjackHand();
         cardsInHand = 2 + (int)(Math.random()*5);
         System.out.println();
         System.out.println();
         System.out.println("Hand contains:");
         for ( int i = 1; i <= cardsInHand; i++ ) {
                // Get a card from the deck, print it out,
                //   and add it to the hand.
            card = deck.dealCard();
            hand.addCard(card);
            System.out.println("    " + card);
         }
         System.out.println("Value of hand is " +
hand.getBlackjackValue());
         System.out.println();
         System.out.print("Again? ");
         again = TextIO.getlnBoolean();
      } while (again == true);

   }

}  // end class TestBlackjackHand
```

Solution for Programming Exercise 5.5

Exercise 5.5:

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.

Discussion

Let's start by designing the main program. We want to give the user $100 for betting on the games. Then, the user plays Blackjack games until the user runs out of money or until the user wants to quit. We could ask the user after each game whether she wants to continue. But instead of this, I just tell the user to enter a bet amount of 0 if she wants to quit. We need variables to represent the amount of money that the user has and the amount that the user bets on a given game. Let `money` and `bet` be variables of type int to represent these quantities. Then, we can write an algorithm for the main program:

```
Let money = 100
while (true):
    Input the user's bet
```

```
    if the bet is 0:
        Break out of the loop
    User plays a game of Blackjack
    if the user won the game
        Pay off the user's bet (money = money + bet)
    else
        Collect the user's bet (money = money - bet)
    If the user is out of money:
        Break out of the loop.
```

Since the Blackjack game will take place in a subroutine, we need some way for the `main()` routine to find out whether the user won. The exercise says that the subroutine should be a function that returns a boolean value with this information. We should record the return value and test it to see whether the user won. The other point that needs some refinement is inputting the user's bet. We better make sure that the user's bet is a reasonable amount, that is, something between 0 and the amount of money the user has. So, the algorithm can be refined as:

```
Let money = 100
while (true):
    do {
        Ask the user to enter a bet
        Let bet = the user's response
    } while bet is < 0 or > money
    if  bet is 0:
        Break out of the loop
    Let userWins = playBlackjack()
    if userWins:
        Pay off the user's bet (money = money + bet)
    else
        Collect the user's bet (money = money - bet)
    If money == 0:
        Break out of the loop.
```

This algorithm can be translated into the `main()` routine in the program given below.

Of course, the major part of the problem is to write the `playBlackjack()` routine. Fortunately, the exercise gives what amounts to a fairly detailed outline of the algorithm. Things are a little complicated because the game can end at various points along the way. When this happens, the subroutine ends

immediately, and any remaining steps in the algorithm are skipped. In outline, the game goes like this:

```
Create and shuffle a deck of cards
Create two BlackjackHands, userHand and dealerHand
Deal two cards into each hand
Check if dealer has blackjack (if so, game ends)
Check if user has blackjack (if so, game ends)
User draws cards (if user goes over 21, game ends)
Dealer draws cards (if dealer goes over 21, games ends)
Check for winner
```

The last three steps need to be expanded, again using the information stated in the exercise. The user can draw several cards, so we need a loop. The loop ends when the user wants to "stand". In the loop, if the value of `userHand` goes over 21, then the whole subroutine ends. The dealer simply draws cards in a loop as long as the value of `dealerHand` is 16 or less. Again, if the value goes over 21, the whole subroutine ends. In the last step, we determine the winner by comparing the values of the two hands. With these refinements, the algorithm becomes

```
Create and shuffle a deck of cards
Create two BlackjackHands, userHand and dealerHand

Deal two cards into each hand
if dealer has blackjack
    User loses and the game ends now
If user has blackjack
    User wins and the game ends now

Repeat:
    Ask whether user wants to hit or stand
    if user stands:
        break out of loop
    if user hits:
        Give user a card
        if userHand.getBlackjackValue() > 21:
            User loses and the game ends now

while  dealerHand.getBlackjackValue() <= 16 :
    Give dealer a card
    if dealerHand.getBlackjackValue() > 21:
        User wins and game ends now
```

```
if dealerHand.getBlackjackValue() >=
userHand.getBlackjackValue()
    User loses
else
    User wins
```

This is ready to be translated into Java. One point of coding is the question of how to deal a card to the user or to the dealer. If `deck` refers to the object of type *Deck*, then the function call `deck.dealCard()` returns the card we want. We can add the card to a hand with the `addCard()` instance method from the *Hand* class. We can do this in one step, if we want. For example, to deal two cards into each hand, we just have to say

```
dealerHand.addCard( deck.dealCard() );
dealerHand.addCard( deck.dealCard() );
userHand.addCard( deck.dealCard() );
userHand.addCard( deck.dealCard() );
```

Of course, a lot of output statements have to be added to the algorithm to keep the user informed about what is going on. For example, I expanded the step where it says "Ask whether user wants to hit or stand" to

```
Display all the cards in the user's hand
Display the user's total
Display the dealers face-up card, i.e.
dealerHand.getCard(0)
Ask if user wants to hit or stand
Get user's response, and make sure it's legal
```

The last step listed here expands to a loop that ends when the user inputs a valid response, 'H' or 'S'. The first step uses a `for` loop to display the cards in the user's hand:

```
for ( int i = 0;  i < userHand.getCardCount();  i++ )
     System.out.println("   " + userHand.getCard(i));
```

The function call `userHand.getCardCount()` gives the number of cards in the hand. The cards are numbered from 0 to `userHand.getCardCount() - 1`, and `userHand.getCard(i)` is the card in the `i`-position. Of course, to produce code like this, you have to make sure that you are familiar with the methods in the classes that you are using.

Although there are many other details to get right, it's mostly routine from here on. I encourage you to read the entire program below and make sure that you understand it.

The Solution

```
import textio.TextIO;

/**
 * This program lets the user play Blackjack.  The computer
 * acts as the dealer.  The user has a stake of $100, and
 * makes a bet on each game.  The user can leave at any
time,
 * or will be kicked out when he loses all the money.
 * House rules:  The dealer hits on a total of 16 or less
 * and stands on a total of 17 or more.  Dealer wins ties.
 * A new deck of cards is used for each game.
 */
public class Blackjack {

   public static void main(String[] args) {

      int money;          // Amount of money the user has.
      int bet;            // Amount user bets on a game.
      boolean userWins;   // Did the user win the game?

      System.out.println("Welcome to the game of
blackjack.");
      System.out.println();

      money = 100;  // User starts with $100.

      while (true) {
          System.out.println("You have " + money + "
dollars.");
          do {
             System.out.println("How many dollars do you
want to bet?  (Enter 0 to end.)");
             System.out.print("? ");
             bet = TextIO.getlnInt();
             if (bet < 0 || bet > money)
                 System.out.println("Your answer must be
between 0 and " + money + '.');
          } while (bet < 0 || bet > money);
          if (bet == 0)
```

```
            break;
         userWins = playBlackjack();
         if (userWins)
            money = money + bet;
         else
            money = money - bet;
         System.out.println();
         if (money == 0) {
            System.out.println("Looks like you've run out
of money!");
            break;
         }
      }

      System.out.println();
      System.out.println("You leave with $" + money + '.');

   } // end main()


   /**
    * Let the user play one game of Blackjack, with the
computer as dealer.
    * @return true if the user wins the game, false if the
user loses.
    */
   static boolean playBlackjack() {

      Deck deck;                  // A deck of cards.  A
new deck for each game.
      BlackjackHand dealerHand;   // The dealer's hand.
      BlackjackHand userHand;     // The user's hand.

      deck = new Deck();
      dealerHand = new BlackjackHand();
      userHand = new BlackjackHand();

      /*  Shuffle the deck, then deal two cards to each
player. */

      deck.shuffle();
      dealerHand.addCard( deck.dealCard() );
      dealerHand.addCard( deck.dealCard() );
      userHand.addCard( deck.dealCard() );
      userHand.addCard( deck.dealCard() );

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

```
      /* Check if one of the players has Blackjack (two
cards totaling to 21).
         The player with Blackjack wins the game.  Dealer
wins ties.
      */

      if (dealerHand.getBlackjackValue() == 21) {
           System.out.println("Dealer has the " +
dealerHand.getCard(0)
                                     + " and the " +
dealerHand.getCard(1) + ".");
           System.out.println("User has the " +
userHand.getCard(0)
                                       + " and the " +
userHand.getCard(1) + ".");
           System.out.println();
           System.out.println("Dealer has Blackjack.
Dealer wins.");
           return false;
      }

      if (userHand.getBlackjackValue() == 21) {
           System.out.println("Dealer has the " +
dealerHand.getCard(0)
                                     + " and the " +
dealerHand.getCard(1) + ".");
           System.out.println("User has the " +
userHand.getCard(0)
                                       + " and the " +
userHand.getCard(1) + ".");
           System.out.println();
           System.out.println("You have Blackjack.  You
win.");
           return true;
      }

      /*  If neither player has Blackjack, play the game.
First the user
          gets a chance to draw cards (i.e., to "Hit").
The while loop ends
          when the user chooses to "Stand".  If the user
goes over 21,
          the user loses immediately.
      */

      while (true) {
```

```
          /* Display user's cards, and let user decide to
Hit or Stand. */

          System.out.println();
          System.out.println();
          System.out.println("Your cards are:");
          for ( int i = 0; i < userHand.getCardCount();
i++ )
             System.out.println("    " +
userHand.getCard(i));
          System.out.println("Your total is " +
userHand.getBlackjackValue());
          System.out.println();
          System.out.println("Dealer is showing the " +
dealerHand.getCard(0));
          System.out.println();
          System.out.print("Hit (H) or Stand (S)? ");
          char userAction;  // User's response, 'H' or
'S'.
          do {
             userAction = Character.toUpperCase(
TextIO.getlnChar() );
             if (userAction != 'H' && userAction != 'S')
                System.out.print("Please respond H or S:
");
          } while (userAction != 'H' && userAction !=
'S');

          /* If the user Hits, the user gets a card.  If
the user Stands,
             the loop ends (and it's the dealer's turn to
draw cards).
          */

          if ( userAction == 'S' ) {
                  // Loop ends; user is done taking cards.
              break;
          }
          else {  // userAction is 'H'.  Give the user a
card.
                  // If the user goes over 21, the user
loses.
              Card newCard = deck.dealCard();
              userHand.addCard(newCard);
              System.out.println();
              System.out.println("User hits.");
              System.out.println("Your card is the " +
newCard);
```

```
                System.out.println("Your total is now " +
userHand.getBlackjackValue());
                if (userHand.getBlackjackValue() > 21) {
                    System.out.println();
                    System.out.println("You busted by going
over 21.  You lose.");
                    System.out.println("Dealer's other card
was the "
                                                            +
dealerHand.getCard(1));
                    return false;
                }
            }

      } // end while loop

      /* If we get to this point, the user has Stood with
21 or less.  Now, it's
         the dealer's chance to draw.  Dealer draws cards
until the dealer's
         total is > 16.  If dealer goes over 21, the dealer
loses.
      */

      System.out.println();
      System.out.println("User stands.");
      System.out.println("Dealer's cards are");
      System.out.println("    " + dealerHand.getCard(0));
      System.out.println("    " + dealerHand.getCard(1));
      while (dealerHand.getBlackjackValue() <= 16) {
         Card newCard = deck.dealCard();
         System.out.println("Dealer hits and gets the " +
newCard);
         dealerHand.addCard(newCard);
         if (dealerHand.getBlackjackValue() > 21) {
            System.out.println();
            System.out.println("Dealer busted by going over
21.  You win.");
            return true;
         }
      }
      System.out.println("Dealer's total is " +
dealerHand.getBlackjackValue());

      /* If we get to this point, both players have 21 or
less.  We
         can determine the winner by comparing the values
of their hands. */
```

```
      System.out.println();
      if (dealerHand.getBlackjackValue() ==
userHand.getBlackjackValue()) {
         System.out.println("Dealer wins on a tie.  You
lose.");
         return false;
      }
      else if (dealerHand.getBlackjackValue() >
userHand.getBlackjackValue()) {
         System.out.println("Dealer wins, " +
dealerHand.getBlackjackValue()
                           + " points to " +
userHand.getBlackjackValue() + ".");
         return false;
      }
      else {
         System.out.println("You win, " +
userHand.getBlackjackValue()
                           + " points to " +
dealerHand.getBlackjackValue() + ".");
         return true;
      }

   }  // end playBlackjack()


} // end class Blackjack
```

Solution for Programming Exercise 5.6

Exercise 5.6:

Exercise 4.8 asked you to write a program that administers a 10-question addition quiz. Rewrite that program so that it uses the following class to represent addition questions:

```
public class AdditionQuestion {

    private int a, b;  // The numbers in the problem.

    public AdditionQuestion() { // constructor
        a = (int)(Math.random() * 50 + 1);
        b = (int)(Math.random() * 50);
    }

    public String getQuestion() {
        return "What is " + a + " + " + b + " ?";
    }

    public int getCorrectAnswer() {
        return a + b;
    }

}
```

Discussion

The original program used two arrays of type int[] to represent the addition questions. One array held the first number from each of the ten problems, while the other array held the second number. This representation is an example of **parallel arrays**: The data for a single question is split between two arrays. (The arrays are called "parallel" because to see all the data for a problem next to each other, you would have to lay the arrays down next to each other, like parallel lines, so the i-th element in one array lies next to the i-th element in the other array.)

The new version of the program can use an array of ten objects of type *AdditionQuestion* to represent the quiz. The data that we need for any one question can now be found in the same place—the object that represents that question. This makes for a neater solution than using parallel arrays.

Some of the functionality that we need to create, administer, and grade the quiz has been incorporated into the *AdditionQuestion* class. For example, the constructor in that class already selects random numbers to be used in the problem, so creating a problem is just a matter of calling the constructor, and there is no need to use random numbers when making a problem in the `createQuiz()` method. Also, to get the question and correct answer for a problem, we can just call `questions[i].getQuestion()` and `questions[i].getCorrectAnswer()`. In fact, we can't get the question any other way, since the two instance variables in an *ArithmeticQuestion* are `private`. One downside of this is that I couln't format the output from `gradeQuiz()` as neatly as in the first version.

All-in-all, it is fairly straightforward to translate the old version of the program into the new version. You can read my solution below.

By the way, I considered adding the user's answer to the *AdditionQuestion* class, along with the correct answer, but the user's answer doesn't seem to me to be logically part of a question. So I think it makes more sense to keep it out of the object that represents the question.

Remember that this program depends on the *AdditionQuestion* class that is given in the statement of the problem.

The Solution

```
import textio.TextIO;

/**
 * This program administers a ten-question addition quiz to
the user.  The numbers
 * for the problem are chosen at random.  The numbers and
the answers are one or
 * two digits.  After asking the user the ten questions,
the computer grades the
```

```
 * quiz, telling the user the correct answer for any
problem they got wrong.
 */
public class AdditionQuizWithObjects {

    private static AdditionQuestion[] questions;  // The
questions for the quiz

    private static int[] userAnswers;   // The user's
answers to the ten questions.


    public static void main(String[] args) {
        System.out.println();
        System.out.println("Welcome to the addition
quiz!");
        System.out.println();
        createQuiz();
        administerQuiz();
        gradeQuiz();
    }


    /**
     * Creates the array of objects that holds the quiz
questions
     */
    private static void createQuiz() {
        questions = new AdditionQuestion[10];
        for ( int i = 0; i < 10; i++ ) {
            questions[i] = new AdditionQuestion();
        }
    }


    /**
     * Asks the user each of the ten quiz questions and
gets the user's answers.
     * The answers are stored in an array, which is created
in this subroutine.
     */
    private static void administerQuiz() {
        userAnswers = new int[10];
        for (int i = 0; i < 10; i++) {
            int questionNum = i + 1;
            System.out.printf("Question %2d:  %s ",
                                    questionNum,
```

```
questions[i].getQuestion());
            userAnswers[i] = TextIO.getlnInt();
        }
    }


    /**
     * Shows all the questions, with their correct answers,
and computes a grade
     * for the quiz.  For each question, the user is told
whether they got
     * it right.
     */
    private static void gradeQuiz() {
        System.out.println();
        System.out.println("Here are the correct
answers:");
        int numberCorrect = 0;
        int grade;
        for (int i = 0; i < 10; i++) {
            int questionNum = i + 1;
            System.out.printf("   Question %2d:  %s
Correct answer is %d  ",
                questionNum, questions[i].getQuestion(),
questions[i].getCorrectAnswer());
            if ( userAnswers[i] ==
questions[i].getCorrectAnswer() ) {
                System.out.println("You were CORRECT.");
                numberCorrect++;
            }
            else {
                System.out.println("You said " +
userAnswers[i] + ", which is INCORRECT.");
            }
        }
        grade = numberCorrect * 10;
        System.out.println();
        System.out.println("You got " + numberCorrect + "
questions correct.");
        System.out.println("Your grade on the quiz is " +
grade);
        System.out.println();
    }

} // end class AdditionQuizWithObjects
```

Solution for Programming Exercise 5.7

Exercise 5.7:

Rewrite the program from the previous exercise so that it administers a quiz with several different kinds of questions. In the previous exercise, you used a class to represent addition questions. For this exercise, you will use the following `interface`, or an equivalent abstract class, to represent the more general idea of a question that has an integer as its answer:

```
public interface IntQuestion {
    public String getQuestion();
    public int getCorrectAnswer();
}
```

You can make the *AdditionQuestion* class implement the interface simply by adding "`implements IntQuestion`" to its definition. Write a similar class to represent subtraction questions. When creating a subtraction problem, you should make sure that the answer is not negative.

For the new program, use an array of type `IntQuestion[]` to hold the quiz questions. Include some addition questions and some subtraction questions in the quiz. You can also add a couple non-math questions, including this one, created as an anonymous class:

```
IntQuestion bigQuestion = new IntQuestion() {
    public String getQuestion() {
        return "What is the answer to the ultimate
question " +
                 " of life, the universe, and
everything?";
    }
    public int getCorrectAnswer() {
        return 42;
    }
};
```

Discussion

Where the solution to the previous exercise used an array of *AdditionQuestion* to store addition problems, the new program has to store questions of various types. An array of type `IntQuestion[]` can hold any object that implements the *IntQuestion* interface. That includes objects of type *AdditionQuestion* as long as we modify that class so that it implements *IntQuestion*. But it can also include other types of objects. For one thing, we want to have subtraction problems. We can implement subtraction with a class very similar to the class for addition questions:

```
class SubtractionQuestion implements IntQuestion {
    private int a, b;  // The numbers in the problem.
    public SubtractionQuestion() { // constructor
       a = (int)(Math.random() * 50 + 1);
       b = (int)(Math.random() * 50);
       if (a - b < 0) { // swap a and b so answer won't be
negative
          int temp = a;
          a = b;
          b = temp;
       }
   }
   public String getQuestion() {
       return "What is " + a + " - " + b + " ?";
    }
    public int getCorrectAnswer() {
       return a - b;
   }
}
```

The exercise asks for subtraction problems for which the answer is not negative. There are a number of ways to accomplish that. For example, you could use a loop that would continue selecting random values for `a` and `b` as long as `a - b < 0`. You could choose *b* and the *answer* randomly and then compute *a* as `a = answer + b`. In my solution, I just swap the values of `a` and `b` when `a - b < 0`.

Creating the quiz becomes a more difficult problem in the new version, since we need to use several different kinds of questions. In my solution, the first seven questions are math problems. For each math problem, the program decides randomly whether to make it an addition question or a subtraction

question. The last three questions are created using anonymous classes, just like the example given in the exercise. You can see my solution below.

But the three anonymous classes in my solution have a lot in common: They are the same except for the particular string that is used to ask the question and the particular number that is the correct answer. A better approach would probably have been to create a class to represent such questions:

```
public class GeneralQuestion implements IntQuestion {
    private String question;
    private int answer;
    public GeneralQuestion( String question, int answer )
{
        this.question = question;
        this.answer = answer;
    }
    public String getQuestion() {
        return question;
    }
    public int getCorrectAnswer() {
        return answer;
    }
}
```

In that case, the subroutine for creating the quiz would become:

```
private static void createQuiz() {
    questions = new IntQuestion[10];
    for ( int i = 0; i < 7; i++ ) {
        if (Math.random() < 0.5)
            questions[i] = new AdditionQuestion();
        else
            questions[i] = new SubtractionQuestion();
    }
    questions[7] = new GeneralQuestion(
                           "How many states are there in the
United States?", 50 );
    questions[8] = new GeneralQuestion(
                           "In what year did the First World
War begin?", 1914 );
    questions[9] = new GeneralQuestion(
                            "What is the answer to the
ultimate question " +
                                  "of life, the universe,
```

```
and everything?", 42 );
}
```

Aside from the declaration of the array `questions` and the routine for creating the quiz, everything else in the program can be exactly as it was in the previous exercise. (However, I did change the format of the output in `gradeQuiz()` to make it look better for longer questions.)

In my solution, just to keep everything in one file, I nested the interface and the two classes that implement it inside the main program class. It would probably be better style to put them all in their own files.

The Solution

```
import textio.TextIO;

/**
 * This program creates, administers, and grades a quiz
made up of ten questions,
 * where each question has an integer answer.  The quiz
includes some simple addition
 * problems, some subtraction problems, and some non-math
questions.
 */
public class GeneralQuiz {

    // -------------------- Nested classes and interface --
---------------------

    interface IntQuestion {
        public String getQuestion();
        public int getCorrectAnswer();
    }

    static class AdditionQuestion implements IntQuestion {
       private int a, b;  // The numbers in the problem.
       public AdditionQuestion() { // constructor
           a = (int)(Math.random() * 50 + 1);
           b = (int)(Math.random() * 50);
       }
       public String getQuestion() {
           return "What is " + a + " + " + b + " ?";
       }
```

```
        public int getCorrectAnswer() {
            return a + b;
        }
    }

    static class SubtractionQuestion implements IntQuestion
{
        private int a, b;  // The numbers in the problem.
        public SubtractionQuestion() { // constructor
            a = (int)(Math.random() * 50 + 1);
            b = (int)(Math.random() * 50);
            if (b > a) { // swap a and b so answer won't be
negative
               int temp = a;
               a = b;
               b = temp;
            }
        }
        public String getQuestion() {
            return "What is " + a + " - " + b + " ?";
        }
        public int getCorrectAnswer() {
            return a - b;
        }
    }

    // -------------------- The Program -------------------
-------------------

    private static IntQuestion[] questions;  // The
questions for the quiz

    private static int[] userAnswers;   // The user's
answers to the ten questions.


    public static void main(String[] args) {
        System.out.println();
        System.out.println("Welcome to the quiz");
        System.out.println();
        System.out.println("There are some math questions
and a few non-math");
        System.out.println("questions, but the answer to
every question is");
        System.out.println("an integer.");
        System.out.println();
        createQuiz();
        administerQuiz();
```

```
        gradeQuiz();
    }


    /**
     * Creates the array of objects that holds the quiz
questions
     */
    private static void createQuiz() {
        questions = new IntQuestion[10];
        for ( int i = 0; i < 7; i++ ) {
            if (Math.random() < 0.5)
                    questions[i] = new AdditionQuestion();
                else
                    questions[i] = new
SubtractionQuestion();
        }
        questions[7] = new IntQuestion() {
              public String getQuestion() {
                  return "How many states are there in the
United States?";
              }
              public int getCorrectAnswer() {
                  return 50;
              }
        };
        questions[8] = new IntQuestion() {
              public String getQuestion() {
                  return "In what year did the First World
War begin?";
              }
              public int getCorrectAnswer() {
                  return 1914;
              }
        };
        questions[9] = new IntQuestion() {
              public String getQuestion() {
                  return "What is the answer to the
ultimate question " +
                                  "of life, the universe, and
everything?";
              }
              public int getCorrectAnswer() {
                  return 42;
              }
        };
   }
```

```
    /**
     * Asks the user each of the ten quiz questions and
gets the user's answers.
     * The answers are stored in an array, which is created
in this subroutine.
     */
    private static void administerQuiz() {
        userAnswers = new int[10];
        for (int i = 0; i < 10; i++) {
            int questionNum = i + 1;
            System.out.printf("Question %2d:  %s ",
                                    questionNum,
questions[i].getQuestion());
            userAnswers[i] = TextIO.getlnInt();
        }
    }


    /**
     * Shows all the questions, with their correct answers,
and computes a grade
     * for the quiz.  For each question, the user is told
whether they got
     * it right.
     */
    private static void gradeQuiz() {
        System.out.println();
        System.out.println("Here are the correct
answers:");
        System.out.println();
        int numberCorrect = 0;
        int grade;
        for (int i = 0; i < 10; i++) {
            System.out.println("Question number " + (i+1) +
":");
            System.out.println("    " +
questions[i].getQuestion());
            System.out.println("    Correct answer:  " +
questions[i].getCorrectAnswer());
            if ( userAnswers[i] ==
questions[i].getCorrectAnswer() ) {
                System.out.println("    You were
CORRECT.");
                numberCorrect++;
            }
            else {
                System.out.println("    You said " +
```

```
userAnswers[i] + ", which is INCORRECT.");
            }
            System.out.println();
        }
        grade = numberCorrect * 10;
        System.out.println();
        System.out.println("You got " + numberCorrect + "
questions correct.");
        System.out.println("Your grade on the quiz is " +
grade);
        System.out.println();
    }

} // end class GeneralQuiz
```

Solution for Programming Exercise 6.1

Exercise 6.1:

In Subsection 6.3.2, the following code was given as an example. It installs a `MousePressed` event handler on a canvas. The handler lets the user draw a red rectangle at the point where the user clicks the mouse, or, by holding the shift key down, a blue oval:

```
canvas.setOnMousePressed( evt -> {
    GraphicsContext g = canvas.getGraphicsContext2D();
    if ( evt.isShiftDown() ) {
        g.setFill( Color.BLUE );
        g.fillOval( evt.getX() - 30, evt.getY() - 15, 60,
30 )
    }
    else {
        g.setFill( Color.RED );
        g.fillRect( evt.getX() - 30, evt.getY() - 15, 60,
30 );
    }
} );
```

Write a complete program that does the same, but in addition, the program will continue to draw figures if the user drags the mouse. That is, the mouse will leave a trail of figures as the user drags. However, if the user right-clicks the canvas, then the canvas should simply be cleared and no figures should be drawn even if the user drags the mouse after right-clicking. See the discussion of dragging in Subsection 6.3.3. Here is a picture of my solution:

Note that a black border has been added around each shape to make them more distinct.

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 where the previous shape was drawn.

Discussion

In order to implement dragging, we need to add a handler for `MouseDragged` events, in addition to the handler for `MousePressed` events. Since the code for handling the events is rather long, I wrote separate methods to handle the events, and the event handlers that I register with the canvas simply call those methods:

```
canvas.setOnMousePressed( e -> mousePressed(e) );
canvas.setOnMouseDragged( e -> mouseDragged(e) );
```

I also decided to use a global variable, `canvasGraphics`, of type *GraphicsContext* to represent the graphics context for drawing on the canvas. I get the graphics context in the `start()` method when I create the canvas, and I fill the canvas with white:

```
Canvas canvas = new Canvas(500,380);
canvasGraphics = canvas.getGraphicsContext2D();
canvasGraphics.setFill(Color.WHITE);
canvasGraphics.fillRect(0,0,500,380);
canvasGraphics.setStroke(Color.BLACK); // stroke color
never changes
```

I wanted to have a black border around the canvas (more for the screenshot than for anything else). I could have drawn the border on the canvas itself, but I decided to apply the border to the pane that is used as the root of the scene graph. This led me to use a *BorderPane* for the root rather than a simple *Pane*. When a *BorderPane* computes its preferred size, it allows space for any border or padding that have been applied to the pane, and the content is correctly placed inside the border and padding. A *Pane* would not take care of that for me. (When I first wrote the program, using a *Pane*, the canvas covered part of the border.) So, the root pane is created with

```
BorderPane root = new BorderPane(canvas);
root.setStyle("-fx-border-color: black; -fx-border-width:
2px");
```

The `mousePressed()` method can use the code given in the exercise, except that the graphics context is named `canvasGraphics` instead of `g`, and code should be added to stroke the shapes with black after filling them. However, before drawing anything, the method must check whether the user right-clicked the canvas; if so, the canvas should be cleared and nothing should be drawn. The test that is needed is if `evt.getButton() == MouseButton.SECONDARY`.

Furthermore, we need to add some global variables to the program to keep track of what is happening as the user drags the mouse. First of all, `mouseDragged()` does not always draw shapes; if the user used the right mouse button, no shapes are drawn. We can implement that with the typical boolean instance variable, `dragging`. This variable is set to `false` in `mousePressed()` if the user right-clicked, and it is set to `true` otherwise. If `dragging` is false in `mouseDragged()`, then that method returns without drawing anything.

In addition, even when `dragging` is `true`, `mouseDragged()` should not draw anything unless the mouse has moved at least 5 pixels since the previous

shape was drawn. We need two more global variables, of type double, to keep track of the mouse position where a shape was most recently drawn. In my program, the variables are `prevShapeX` and `prevShapeY`. The variables' values should be set every time a shape is drawn, in either `mousePressed()` or `mouseDragged()`, and `mouseDragged()` should compare their values to the current mouse location to decide whether the mouse has moved far enough to draw the next shape. It is not completely clear what "move at least 5 pixels" means, but I take it to mean "move at least 5 pixels vertically or at least 5 pixels horizontally". So, `mouseDragged()` starts off like this:

```
public void mouseDragged(MouseEvent evt) {
    if ( dragging == false ) {
        return;
    }

    double x = evt.getX();  // x-coordinate where user
clicked.
    double y = evt.getY();  // y-coordinate where user
clicked.

    if ( Math.abs(x - prevShapeX) < 5 && Math.abs(y -
prevShapeY) < 5 ) {
            // The mouse has not moved at least 5 pixels
horizontally
            // or vertically, so don't draw another shape
yet.
        return;
    }

    prevShapeX = x;  // Save coords where the next shape
is being drawn.
    prevShapeY = y;
```

The Solution

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.paint.Color;
import javafx.scene.layout.BorderPane;
```

```
import javafx.scene.input.MouseEvent;
import javafx.scene.input.MouseButton;


/**
 * A simple demonstration of MouseEvents.  Shapes are drawn
 * on a white background when the user clicks the canvas.
If
 * the user right-clicks, the canvas is cleared.  If the
user
 * shift-clicks the canvas, a blue oval is drawn.
Otherwise,
 * when the user clicks, a red rectangle is drawn.  If the
user
 * did not right-click, then ovals or rects continue to be
 * drawn as the user drags the mouse.
 */
public class SimpleStamperWithDrag extends Application {

    public static void main(String[] args) {
        launch();
    }

    // ---------------------------------------------------
------------------

    /**
     * This variable is set to true during a drag
operation, unless the
     * user was holding down the shift key when the mouse
was first
     * pressed (since in that case, the mouse gesture
simply clears the
     * canvas and no figures should be drawn if the user
drags the mouse).
     */
    private boolean dragging;

    /**
     * While dragging, prevShapeX and prevShapeY are the
coordinates
     * at which the previous shape was drawn.  They are
used to avoid
     * drawing the next shape until the mouse has moved at
least 5 pixels
     * horizontally or vertically.
     */
    private double prevShapeX, prevShapeY;
```

```
    /**
     * A graphics context for drawing on the canvas that
fills the screen.
     */
    private GraphicsContext canvasGraphics;


    /**
     * This start() method sets up the GUI to show a canvas
where the shapes
     * are drawn, and it installs mouse handlers on the
canvas to draw shapes
     * as the user presses and drags the mouse.
     */
    public void start(Stage stage) {

        Canvas canvas = new Canvas(500,380);
        canvasGraphics = canvas.getGraphicsContext2D();
        canvasGraphics.setFill(Color.WHITE);
        canvasGraphics.fillRect(0,0,500,380);
        canvasGraphics.setStroke(Color.BLACK); // stroke
color never changes

        canvas.setOnMousePressed( e -> mousePressed(e) );
        canvas.setOnMouseDragged( e -> mouseDragged(e) );

        BorderPane root = new BorderPane(canvas);
        root.setStyle("-fx-border-color: black; -fx-border-
width: 2px");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Mouse Drag Demo");
        stage.setResizable(false);
        stage.show();

    } // end start()


    /**
     *  This method will be called when the user clicks the
mouse on the canvas.
     *  If the user right-clicked, it clears the canvas.
Otherwise, it draws
     *  a shape and starts a drag operation, but only if
the mouse has moved
     *  sufficiently far since drawing the previous shape.
```

```
     */
    public void mousePressed(MouseEvent evt) {

        if ( evt.getButton() == MouseButton.SECONDARY ) {
                // The user right-clicked the canvas.  Fill
the canvas with white
                // to erase its current contents.
            dragging = false;
            canvasGraphics.setFill(Color.WHITE);
            canvasGraphics.fillRect(0,0,500,380);
            return;
        }

        dragging = true;

        double x = evt.getX();  // x-coordinate where user
clicked.
        double y = evt.getY();  // y-coordinate where user
clicked.

        prevShapeX = x;  // Save coordinates where first
shape is drawn.
        prevShapeY = y;

        if ( evt.isShiftDown() ) {
                // User was holding down the shift key.
Draw a blue oval centered
                // at the point (x,y). (A black outline
around the oval will make it
                // more distinct when shapes overlap.)
            canvasGraphics.setFill(Color.BLUE);  // Blue
interior.
            canvasGraphics.fillOval( x - 30, y - 15, 60, 30
);
            canvasGraphics.strokeOval( x - 30, y - 15, 60,
30 );
        }
        else {
                // Draw a red rectangle centered at (x,y).
            canvasGraphics.setFill(Color.RED);   // Red
interior.
            canvasGraphics.fillRect( x - 30, y - 15, 60, 30
);
            canvasGraphics.strokeRect( x - 30, y - 15, 60,
30 );
        }

    } // end mousePressed();
```

```
    /**
     *  This method is called when the user drags the
mouse.  If a the value of the
     *  instance variable dragging is true, it will draw a
rect or oval at the
     *  current mouse position.
     */
    public void mouseDragged(MouseEvent evt) {
        if ( dragging == false ) {
            return;
        }

        double x = evt.getX();  // x-coordinate where user
clicked.
        double y = evt.getY();  // y-coordinate where user
clicked.

        if ( Math.abs(x - prevShapeX) < 5 && Math.abs(y -
prevShapeY) < 5 ) {
                // The mouse has not moved at least 5
pixels horizontally
                // or vertically, so don't draw another
shape yet.
            return;
        }

        prevShapeX = x;  // Save coords where the next
shape is being drawn.
        prevShapeY = y;

        if (evt.isShiftDown() ) {
                // User was holding down the shift key.
Draw a blue oval centered
                // at the point (x,y). (A black outline
around the oval will make it
                // more distinct when shapes overlap.)
            canvasGraphics.setFill(Color.BLUE);  // Blue
interior.
            canvasGraphics.fillOval( x - 30, y - 15, 60, 30
);
            canvasGraphics.strokeOval( x - 30, y - 15, 60,
30 );
        }
        else {
                // Draw a red rectangle centered at (x,y).
            canvasGraphics.setFill(Color.RED);   // Red
```

```
interior.
            canvasGraphics.fillRect( x - 30, y - 15, 60, 30
);
            canvasGraphics.strokeRect( x - 30, y - 15, 60,
30 );
        }

    } // end mouseDragged();


} // end class SimpleStamperWithDrag
```

Solution for Programming Exercise 6.2

Exercise 6.2:

Write a program 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 out of the window if she wants, and it will disappear. To allow the user to get the squares back into the window, add a `KeyPressed` event handler that will restore the squares to their original positions when the user presses the Escape key. The key code for the Escape key is `KeyCode.ESCAPE`.

Discussion

Since the squares can move, the program needs global variables to keep track of the current location of each square. I do that with instance variables `x1` and `y1` for the upper left corner of the red square and `x2` and `y2` for the upper left corner of the blue square. These variables are of type double since mouse positions are given as real numbers in JavaFX. The canvas is also a global variable. I wrote a method, `draw()`, to completely redraw the content of the canvas. It is called in the `start()` method and whenever one of the squares is moved. The `draw()` method can get the graphics context that it needs to do its drawing from the canvas:

```
private void draw() {
    GraphicsContext g = canvas.getGraphicsContext2D();
    g.setFill(Color.rgb(230,255,230)); // light green
    g.fillRect(0,0,canvas.getWidth(),canvas.getHeight());
    g.setFill(Color.RED);
    g.fillRect(x1, y1, 30, 30);
    g.setFill(Color.BLUE);
    g.fillRect(x2, y2, 30, 30);
}
```

To implement dragging, the program needs event handlers for `MousePressed`, `MouseDragged`, and (maybe) `MouseReleased` events. The handlers need to be added to the canvas that fills the window. My event handlers simply call methods such as `mousePressed(evt)` that are defined elsewhere in the program.

For this simple application of dragging, the program doesn't need to keep track of the previous or starting positions of the mouse. But as usual for dragging, I use a boolean variable, `dragging`, to keep track of whether or not a drag operation is in progress. Not every mouse press starts a drag operation. If the user clicks the canvas outside of the squares, there is nothing to drag, and so `dragging` is set to false in the `mousePressed` method. If the user clicks one of the squares, `dragging` is set to true. Since there are two squares to be dragged, we have to keep track of which is being dragged. I use a boolean variable, `dragRedSquare`, which is true if the red square is being dragged and is false if the blue square is being dragged. (A boolean variable is actually not the best choice in this case. It would be a problem if we wanted to add another square. A boolean variable only has two possible values, so an integer variable would probably be a better choice. Better yet, the rectangles could be represented by objects, and a the global variable would refer to the object that is being dragged, or would be `null` when no object is being dragged.)

There is one little problem. The mouse location is a single `(x,y)` point. A square occupies a whole bunch of points. When we move the square to follow the mouse, where exactly should we put the square? One possibility is to put the upper left corner of the square at the mouse location. If we did this, the `mouseDragged` routine would look like:

```
public void mouseDragged(MouseEvent evt) {
    if (dragging == false)
      return;
    double x = evt.getX();  // Get mouse position.
    double y = evt.getY();
    if (dragRedSquare) {  // Move the red square.
       x1 = x;  // Put top-left corner at mouse position.
       y1 = y;
    }
    else {   // Move the blue square.
       x2 = x;  // Put top-left corner at mouse position.
```

```
        y2 = y;
    }
    draw();  // redraw canvas with square in new location
}
```

This works, but it is not very aesthetic. When the user starts dragging a square, no matter where in the square the user clicks, the square will jump so that its top-left corner is at the mouse position. This is not what a user typically expects. If I grab a square by clicking its center, then I want the center to stay under the mouse cursor as I move it. If I grab the lower right corner, I want the lower right corner to follow the mouse, not the upper left corner. There is a solution to this, and it's one that is often needed for dragging operations. We need to record the original position of the mouse relative to the upper left corner of the square. This tells us where in the square the user clicked. This is done in the `mousePressed()` method by assigning appropriate values to two new instance variables `offsetX` and `offsetY`:

```
double x = evt.getX();  // Location where user clicked.
double y = evt.getY();

if (x >= x2 && x < x2+30 && y >= y2 && y < y2+30) {
      // It's the blue square (which should be checked
first,
      // since it's drawn on top of the red square.)
   dragging = true;
   dragRedSquare = false;
   offsetX = x - x2;  // Distance from corner of square to
(x,y).
   offsetY = y - y2;
}
else if (x >= x1 && x < x1+30 && y >= y1 && y < y1+30) {
      // It's the red square.
   dragging = true;
   dragRedSquare = true;
   offsetX = x - x1;  // Distance from corner of square to
(x,y).
   offsetY = y - y1;
}
```

In `mouseDragged()`, when the mouse moves to a new `(x,y)` point, we move the square so that the vertical and horizontal distances between the mouse

location and the top left corner of the square remain the same, as given by `offsetX` and `offsetY`:

```
if (dragRedSquare) {  // Move the red square.
   x1 = x - offsetX;  // Offset corner from mouse
location.
   y1 = y - offsetY;
}
else {   // Move the blue square.
   x2 = x - offsetX;  // Offset corner from mouse
location.
   y2 = y - offsetY;
}
```

An alternative to using `offsetX` and `offsetY` would be to keep track of `prevX` and `prevY`, the previous values of `x` and `y`. Then, in `mouseDragged()`, you can use the current and previous values of `x` and `y` to determine how far the mouse has moved. Then, you can simply move the square by the same amount. This idea can be implemented in `mouseDragged()` with the following code:

```
int x = evt.getX();  // Current location of the mouse.
int y = evt.getY();

int dx = x - prevX;   // amount by which the mouse has
moved.
int dy = y - prevY;

if (dragRedSquare) {
   x1 = x1 + dx;  // move the red square by the same
amount
   y1 = y1 + dy;
}
else {
   x2 = x2 + dx;  // move the blue square by the same
amount
   y2 = y2 + dy;
}

prevX = x;   // new values of prevX,prevY for next call to
mouseDragged()
prevY = y;
```

By the way, if you wanted to stop the user from dragging the square outside the window, you would just have to add code to the `mouseDragged` routine to "clamp" the variables `x1`, `y1`, `x2`, and `y2` so that they lie in the acceptable range. Here is a modified routine that keeps the square entirely within the canvas:

```
public void mouseDragged(MouseEvent evt) {
    if (dragging == false)
      return;
    double x = evt.getX();
    double y = evt.getY();
    if (dragRedSquare) {  // Move the red square.
       x1 = x - offsetX;
       y1 = y - offsetY;
       if (x1 < 0)  // Clamp (x1,y1) so the square lies in
the canvas.
          x1 = 0;
       else if (x1 >= canvas.getWidth() - 30)
          x1 = canvas.getWidth() - 30;
       if (y1 < 0)
          y1 = 0;
       else if (y1 >= canvas.getHeight() - 30)
          y1 = canvas.getHeight() - 30;
    }
    else {   // Move the blue square.
       x2 = x - offsetX;
       y2 = y - offsetY;
       if (x2 < 0)  // Clamp (x2,y2) so the square lies in
the canvas.
          x2 = 0;
       else if (x2 >= canvas.getWidth() - 30)
          x2 = canvas.getWidth() - 30;
       if (y2 < 0)
          y2 = 0;
       else if (y2 >= canvas.getHeight() - 30)
          y2 = canvas.getHeight() - 30;
    }
    draw();
}
```

Finally, we need to program a response to the Escape key. As discussed in Subsection 6.3.4, we can register the `KeyPressed` event handler with the scene. The handler simply has to reset the variables `x1`, `y1`, `x2`, and `y2` to their original values and redraw the canvas. I define the handler in the `start()` method after creating the `scene`:

```
Scene scene = new Scene(root);
scene.setOnKeyPressed( e -> {
       // If user pressed ESCAPE, move squares
       // back to starting positions, and redraw.
    if ( e.getCode() == KeyCode.ESCAPE ) {
        x1 = 10;
        y1 = 10;
        x2 = 50;
        y2 = 10;
        draw();
    }
});
```

The Solution

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.input.KeyCode;
import javafx.scene.input.MouseEvent;

/**
 * A program that shows a red square and a blue square that
the user
 * can drag with the mouse.   The user can drag the squares
off
 * the canvas and drop them.  Pressing the escape key will
restore
 * both squares to their original positions.
 */
public class DragTwoSquares extends Application {

    public static void main(String[] args) {
        launch(args);
    }

    //-------------------------------------------------------
----------------


    private double x1 = 10, y1 = 10;   // Coords of top-
left corner of the red square.
```

```
    private double x2 = 50, y2 = 10;   // Coords of top-
left corner of the blue square.

    private Canvas canvas;  // The canvas where the sqaures
are drawn.


    /**
     *  The start method sets up the GUI.  It adds mouse
event handlers to
     *  the canvas to implement dragging.  It adds a key
pressed handler
     *  to the scene that will restore the squares to their
original
     *  positions when the user presses the escape key.
     */
    public void start(Stage stage) {

        canvas = new Canvas(300,250);
        draw(); // show squares in original positions

        canvas.setOnMousePressed( e -> mousePressed(e) );
        canvas.setOnMouseDragged( e -> mouseDragged(e) );
        canvas.setOnMouseReleased( e -> mouseReleased(e) );

        Pane root = new Pane(canvas);

        Scene scene = new Scene(root);

        scene.setOnKeyPressed( e -> {
               // If user pressed ESCAPE, move squares
               // back to starting positions, and redraw.
            if ( e.getCode() == KeyCode.ESCAPE ) {
                x1 = 10;
                y1 = 10;
                x2 = 50;
                y2 = 10;
                draw();
            }
        });

        stage.setScene(scene);
        stage.setTitle("Drag the squares!");
        stage.setResizable(false);
        stage.show();
    }
```

```
    /**
     * Draw the canvas, showing the squares in their
current positions.
     */
    private void draw() {
        GraphicsContext g = canvas.getGraphicsContext2D();
        g.setFill(Color.rgb(230,255,230)); // light green

g.fillRect(0,0,canvas.getWidth(),canvas.getHeight());
        g.setFill(Color.RED);
        g.fillRect(x1, y1, 30, 30);
        g.setFill(Color.BLUE);
        g.fillRect(x2, y2, 30, 30);
    }


    //-----------------  Variables and methods for
responding to drags -----------

    private boolean dragging;      // Set to true when a
drag is in progress.

    private boolean dragRedSquare; // True if red square is
being dragged, false
                                   //    if blue square is
being dragged.

    private double offsetX, offsetY;  // Offset of mouse-
click coordinates from the
                                      //   top-left corner
of the square that was
                                      //   clicked.

    /**
     * Respond when the user presses the mouse on the
canvas.
     * Check which square the user clicked, if any, and
start
     * dragging that square.
     */
    public void mousePressed(MouseEvent evt) {

        if (dragging)  // Exit if a drag is already in
progress.
            return;

        double x = evt.getX();  // Location where user
clicked.
```

```
        double y = evt.getY();

        if (x >= x2 && x < x2+30 && y >= y2 && y < y2+30) {
                // It's the blue square (which should be
checked first,
                // since it's drawn on top of the red
square.)
            dragging = true;
            dragRedSquare = false;
            offsetX = x - x2;  // Distance from corner of
square to (x,y).
            offsetY = y - y2;
        }
        else if (x >= x1 && x < x1+30 && y >= y1 && y <
y1+30) {
                // It's the red square.
            dragging = true;
            dragRedSquare = true;
            offsetX = x - x1;  // Distance from corner of
square to (x,y).
            offsetY = y - y1;
        }

    }

    /**
     * Dragging stops when user releases the mouse button.
     */
    public void mouseReleased(MouseEvent evt) {
        dragging = false;
    }

    /**
     * Respond when the user drags the mouse.  If a square
is
     * not being dragged, then exit. Otherwise, change the
position
     * of the square that is being dragged to match the
position
     * of the mouse.  Note that the corner of the square is
placed
     * in the same relative position with respect to the
mouse that it
     * had when the user started dragging it.
     */
    public void mouseDragged(MouseEvent evt) {
        if (dragging == false)
            return;
```

```
        double x = evt.getX();
        double y = evt.getY();
        if (dragRedSquare) {  // Move the red square.
            x1 = x - offsetX;
            y1 = y - offsetY;
        }
        else {   // Move the blue square.
            x2 = x - offsetX;
            y2 = y - offsetY;
        }
        draw();  // (Calls the draw() to show squares in
new positions.)
    }

} // end class DragTwoSquares
```

Solution for Programming Exercise 6.3

Exercise 6.3:

Write a program that shows a pair of dice. The dice are drawn on a *Canvas*. You can assume that the size of the canvas is 100 by 100 pixels. When the user clicks on the canvas, 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, such as "`void drawDie(GraphicsContext g, int val, int x, int y)`", to draw a die at the specified `(x,y)` coordinates. The second parameter, `val`, specifies the number of dots that are showing on the die. Here is a picture of a canvas displaying two the dice:

Discussion

This is largely an exercise in precision drawing.

We will need a *Canvas* for drawing the dice. In my program, it is a global variable. We need to add a `MousePressed` event handler to the canvas. My handler calls a method, `roll()`, that will assign new values to the dice and redraw the canvas. The code is in the `start()` method:

```
canvas = new Canvas(100,100);  // Canvas is 100-by-100
pixels.
draw();  // Draw the original dice.
canvas.setOnMousePressed( e -> roll() );  // Add the event
handler.
```

The `draw()` method completely redraws the canvas. The hardest part of this exercise was drawing the dice. I made each die 35 pixels wide, leaving a 10 pixel border on each side and 10 pixels between the dice, for a total of 100 pixels. The top left corner of the left die is at `(10,10)`, the top left corner of the right die is at `(55,55)`. The 55 includes the 10 pixel border on the left, the 35 pixel width of the other die, and the 10 pixels between the dice. The `paintComponent()` method calls a `drawDie()` routine to draw each die, using the commands:

```
drawDie(g, die1, 10, 10);
drawDie(g, die2, 55, 55);
```

where `die1` is the numerical value shown on the first die and `die2` is the numerical value of the second die.

As for the `drawDie` routine, there are two quite different algorithms that could have been used for drawing the dots. Either:

```
if the value shown is 1
    draw 1 dot (in the center)
else if the value shown is 2
    draw 2 dots (in the top-left and bottom-right corners)
  .
  .
  .
else if the value shown is 6
    draw 6 dots (along the left and right edges)
```

Or:

```
if the value has a dot in the top-left corner
     draw the top-left dot
else if the value has a dot in the top-right corner
     draw the top-right dot
   .
   .
   .
else if the value has a dot in the bottom-right corner
     draw the bottom-right dot
```

Although the first algorithm is more obvious, the second requires much less typing. (The first algorithm ends up using 21 `fillOval()` commands, while

the second uses only 7.) Furthermore, after drawing the dice on paper, I found that the conditions for testing when a given dot needs to be drawn are simpler than I expected. For example, the values that need a dot in the top-left position are all the values greater than 1. The algorithm leads to my `drawDie()` routine:

```
/**
 * Draw a die with upper left corner at (x,y).  The die is
 * 35 by 35 pixels in size.  The val parameter gives the
 * value showing on the die (that is, the number of dots).
 */
private void drawDie(GraphicsContext g, int val, int x,
int y) {
    g.setFill(Color.WHITE);
    g.fillRect(x, y, 35, 35);
    g.setStroke(Color.BLACK);
    g.strokeRect(x+0.5, y+0.5, 34, 34);
    g.setFill(Color.BLACK);
    if (val > 1)  // upper left dot
       g.fillOval(x+3, y+3, 9, 9);
    if (val > 3)  // upper right dot
       g.fillOval(x+23, y+3, 9, 9);
    if (val == 6) // middle left dot
       g.fillOval(x+3, y+13, 9, 9);
    if (val % 2 == 1) // middle dot (for odd-numbered
val's)
       g.fillOval(x+13, y+13, 9, 9);
    if (val == 6) // middle right dot
       g.fillOval(x+23, y+13, 9, 9);
    if (val > 3)  // bottom left dot
       g.fillOval(x+3, y+23, 9, 9);
    if (val > 1)  // bottom right dot
       g.fillOval(x+23, y+23, 9,9);
}
```

It took some care to figure out the numbers to use in the `fillOval` commands. The individual dots have a diameter of 9 pixels. There are three rows of dots, which have a combined height of 27 pixels. That leaves 35 minus 27, or 8 pixels for spacing. I use 3 pixels between the dots and the edge of the die, and 1 pixel between rows. This puts the tops of the rows at 3, 3+9+1, and 3+9+1+9+1, that is, at 3, 13, and 23. The columns use the same numbers. (If you believe that I got all this right the first time, I won't disillusion you!)

The `draw()` method fills the canvas with light blue, draws a 2-pixel blue border around the canvas, and draws the two dice:

```
public void draw()) {
    GraphicsContext g = canvas.getGraphicsContext2D();
    g.setFill(Color.rgb(200,200,255));
    g.fillRect(0,0,100,100); // fill canvas with light
blue
    g.setLineWidth(2);
    g.strokeRect(1,1,98,98); // draw a blue border
    drawDie(g, die1, 10, 10);  // draw the dice
    drawDie(g, die2, 55, 55);
}
```

Note the different parameters to `g.fillRect()` and `g.strokeRect()`. To cover the entire canvas, we need to fill a 100-by-100 rectangle, with its top-left corner at (0,0), so the command is `g.fillRect(0,0,100,100)`. But when a rectangle is stroked, the **center** of the pen is dragged along the boundary of the rectangle. For a 2-pixel border, we need to stroke a rectangle that is inset from the boundary of the canvas by half the size of the pen, that is, by one pixel. Since the rectangle is inset by 1 pixel on each side, it is 2 pixels smaller than the full 100-pixel canvas, hence its width and height are 98. If I had said `g.strokeRect(0,0,100,100)`, `half of the pen would lie outside the canvas, and we would only see a 1-pixel border.`

The Solution

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;

/**
 * Shows a pair of dice that are rolled when the user
clicks on the
 * canvas where the dice are drawn.
 */
public class RollDice extends Application {
```

```
    public static void main(String[] args) {
        launch();
    }

    //-------------------------------------------------------
----------------

    private int die1 = 4;  // The values shown on the dice.
    private int die2 = 3;

    private Canvas canvas;  // The canvas on which the dice
are drawn.


    /**
     *  The start() method sets up the GUI and installs a
mouse listener
     *  on the canvas where the dice are to be drawn.
     */
    public void start(Stage stage) {

        canvas = new Canvas(100,100);
        draw();  // Draw the original dice.

        canvas.setOnMousePressed( e -> roll() );

        Pane root = new Pane(canvas);
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Dice!");
        stage.setResizable(false);
        stage.show();

    }


    /**
     * Draw a die with upper left corner at (x,y).  The die
is
     * 35 by 35 pixels in size.  The val parameter gives
the
     * value showing on the die (that is, the number of
dots).
     */
    private void drawDie(GraphicsContext g, int val, int x,
int y) {
        g.setFill(Color.WHITE);
```

```
        g.fillRect(x, y, 35, 35);
        g.setStroke(Color.BLACK);
        g.strokeRect(x+0.5, y+0.5, 34, 34);
        g.setFill(Color.BLACK);
        if (val > 1)  // upper left dot
            g.fillOval(x+3, y+3, 9, 9);
        if (val > 3)  // upper right dot
            g.fillOval(x+23, y+3, 9, 9);
        if (val == 6) // middle left dot
            g.fillOval(x+3, y+13, 9, 9);
        if (val % 2 == 1) // middle dot (for odd-numbered
val's)
            g.fillOval(x+13, y+13, 9, 9);
        if (val == 6) // middle right dot
            g.fillOval(x+23, y+13, 9, 9);
        if (val > 3)  // bottom left dot
            g.fillOval(x+3, y+23, 9, 9);
        if (val > 1)  // bottom right dot
            g.fillOval(x+23, y+23, 9,9);
    }


    /**
     * Roll the dice by randomizing their values.  Tell the
     * system to repaint the canvas, to show the new
values.
     */
    private void roll() {
        die1 = (int)(Math.random()*6) + 1;
        die2 = (int)(Math.random()*6) + 1;
        draw();
    }


    /**
     * The draw() method just draws the two dice and draws
     * a two-pixel wide blue border around the canvas.
     */
    private void draw() {
        GraphicsContext g = canvas.getGraphicsContext2D();
        g.setFill(Color.rgb(200,200,255));
        g.fillRect(0,0,100,100);
        g.setStroke( Color.BLUE );
        g.setLineWidth(2);
        g.strokeRect(1,1,98,98);
        drawDie(g, die1, 10, 10);
        drawDie(g, die2, 55, 55);
    }
```

```
} // end class RollDice
```

Solution for Programming Exercise 6.4

Exercise 6.4:

In Exercise 6.3, you wrote a graphical pair-of-dice program where the dice are rolled when the user clicks on the canvas. Now make a pair-of-dice program where the user rolls the dice by clicking a button. The button should appear under the canvas 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.

Discussion

In Exercise 6.3, the entire window was filled by a *Canvas*. In the new version, we need both a canvas and a button. We need to add both components to the container that acts as the root of the scene graph. I decided to use a `BorderPane` with the canvas in the center position and the button in the bottom position. (Another option would have been a *VBox*.) When I first tried this, the button only filled the left half of the bottom position. A `BorderPane` wants to expand the components that it contains to completely fill their positions in the layout, but as mentioned in Section 6.5, the problem is that a *Button* has a maximum size that is just large enough to hold the contents of the button. The problem can be fixed by giving the button a bigger maximum width. After I did that, the button filled the entire width of the window. Here is the new `start()` method:

```
public void start(Stage stage) {

    canvas = new Canvas(100,100);
    draw();  // Draw the original dice.

    rollButton = new Button("Roll!");
    rollButton.setMaxWidth(1000);  // so button can grow
```

```
to full width of window
    rollButton.setOnAction( e -> roll() ); // When
clicked, roll the dice.

    BorderPane root = new BorderPane();
    root.setCenter(canvas);
    root.setBottom(rollButton);

    Scene scene = new Scene(root);
    stage.setScene(scene);
    stage.setTitle("Dice!");
    stage.setResizable(false);
    stage.show();

} // end start()
```

The method for drawing the dice was discussed in the solution to Exercise 6.3. But we still have to think about animating the rolling of the dice. The `roll()` method is responsible for rolling the dice. In the original version, this method simply set the numbers showing on the dice to random values and called `draw()`. In the new version, we want to repeat this action several times over a period of time. That is, we want an animation in which the action for each frame is to randomize the numbers on the dice and call `draw()`.

Animation is discussed in Subsection 6.3.5. We need to make a subclass of *AnimationTimer* and provide a definition for the `handle()` method. That method will be called once in each frame. The timer can be defined using an anonymous nested class (see Subsection 5.8.3.) In my program, the timer is a global variable. It might seem like it would be enough to say:

```
private AnimationTimer timer = new AnimationTimer() {
    public void handle( long time ) {
        die1 = (int)(Math.random()*6) + 1;
        die2 = (int)(Math.random()*6) + 1;
        draw();
    }
};
```

Then, when the `roll()` method is called, it could simply call `time.start()`. But there is a big problem with this: The animation should only go on for a short time. In order to stop it, `timer.stop()` has to be

called. But who will call it, and how will they know when to call it? In fact, the timer can stop itself after a certain amount of time has passed. In order to do that, I use a global `startTime` variable. When `roll()` starts the timer, it sets the start time to the current time in nanoseconds:

```
private void roll() {
   rollButton.setDisable(true);
   startTime = System.nanoTime();
   timer.start(); // start an animation
}
```

Note that it also disables the button, so that the user will not be able to roll the dice while they are already being rolled. Then the timer's `handle()` method should check the difference between the current time and the start time, and it should stop itself if the time has reached some maximum value. The current time, in nanoseconds, is passed as a parameter to the `handle()` method. My program stops the animation after one second (1,000,000,000 nanoseconds) has passed. The method must also re-enable the button to make it possible to roll the dice again. So, here is the actual definition of the timer:

```
private AnimationTimer timer = new AnimationTimer() {
        // The timer is used to animate "rolling" of the
dice.
        // In each frame, the dice values are randomized.
When
        // the elapsed time reaches 1 second, the timer
stops itself.
        // The rollButton is disabled while an animation
is in
        // progress, so it has to be enabled when the
animation stops.
    public void handle( long time ) {
        die1 = (int)(Math.random()*6) + 1;
        die2 = (int)(Math.random()*6) + 1;
        draw();
        if ( time - startTime >= 1_000_000_000 ) {
            timer.stop();
            rollButton.setDisable(false);
        }
    }
};
```

You might want to slow down the rate at which the numbers on the dice change. Unfortunately, there is no way to slow down the frame rate in an *AnimationTimer*. However, it is certainly possible to do nothing in some of the frames. To implement that, you could record the time when the dice are drawn, and not draw them again until a certain time has passed. For example, if you want to draw the dice every 1/10 second, The `handle()` method could be written:

```
public void handle( long time ) {
    if ( time - previousDrawTime > 100_000_000 ) {
        previousDrawTime = time;
        die1 = (int)(Math.random()*6) + 1;
        die2 = (int)(Math.random()*6) + 1;
        draw();
        if ( time - startTime >= 1_000_000_000 ) {
            timer.stop();
            rollButton.setDisable(false);
        }
    }
}
```

Here, `previousDrawTime` is a global variable. The test "`if (time-previousDrawTime > 100_000_000)`" will always be true the first time `handle()` is called, every time the button is clicked. You might think about why that is true.

The Solution

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.layout.BorderPane;
import javafx.scene.control.Button;
import javafx.scene.paint.Color;
import javafx.animation.AnimationTimer;


/**
 * Shows a pair of dice that are rolled when the user
clicks a button
```

```
 * that appears below the dice.  A short animation runs
when the
 * button is clicked, showing the numbers on the dice
changing.
 */
public class RollDiceWithButton extends Application {

    public static void main(String[] args) {
        launch();
    }

    //-------------------------------------------------------
----------------

   private int die1 = 4;  // The values shown on the dice.
   private int die2 = 3;

   private Canvas canvas; // The canvas where the dice are
drawn

   private Button rollButton;  // The button that is
clicked to roll the dice.

   private long elapsedTime;  // When an animation is
running, the number of
                              //    nanoseconds for which
it has been running.  This
                              //    is used to end the
animation after 1 second.
                              //    (One second is
1,000,000,000 nanoseconds.

   private long startTime;   // Time, in nanoseconds, when
the animaion started.

   private AnimationTimer timer = new AnimationTimer() {
           // The timer is used to animate "rolling" of the
dice.
           // In each frame, the dice values are
randomized.  When
           // the elapsed time reaches 1 second, the timer
stops itself.
           // The rollButton is disabled while an animation
is in
           // progress, so it has to be enabled when the
animation stops.
       public void handle( long time ) {
           die1 = (int)(Math.random()*6) + 1;
```

```
            die2 = (int)(Math.random()*6) + 1;
            draw();
            if ( time - startTime >= 1_000_000_000 ) {
                timer.stop();
                rollButton.setDisable(false);
            }
        }
    };


    /**
     *  The start() method sets up the GUI, using a
BorderPane in which
     *  the canvas is the center component and the button is
the bottom
     *  component.  An ActionEvent handler is added to the
button to
     *  roll the dice when the button is clicked.
     */
    public void start(Stage stage) {

        canvas = new Canvas(100,100);
        draw();  // Draw the original dice.

        rollButton = new Button("Roll!");
        rollButton.setMaxWidth(1000);  // so button can grow
to full width of window
        rollButton.setOnAction( e -> roll() ); // When
clicked, roll the dice.

        BorderPane root = new BorderPane();
        root.setCenter(canvas);
        root.setBottom(rollButton);

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Dice!");
        stage.setResizable(false);
        stage.show();

    } // end start()


    /**
     * Roll the dice by starting an animation that
randomizes
     * the values on the dice in each frame.  The animation
will
```

```
    * last for one second, and the rollButton is disabled
while
    * the animation is in progress.  This method is called
    * when the user clicks the roll button.
    */
   private void roll() {
       rollButton.setDisable(true);
       startTime = System.nanoTime();
       timer.start(); // start an animation
   }


  /**
    * Draw a die with upper left corner at (x,y).  The die
is
    * 35 by 35 pixels in size.  The val parameter gives the
    * value showing on the die (that is, the number of
dots).
    */
   private void drawDie(GraphicsContext g, int val, int x,
int y) {
       g.setFill(Color.WHITE);
       g.fillRect(x, y, 35, 35);
       g.setStroke(Color.BLACK);
       g.strokeRect(x+0.5, y+0.5, 34, 34);
       g.setFill(Color.BLACK);
       if (val > 1)  // upper left dot
           g.fillOval(x+3, y+3, 9, 9);
       if (val > 3)  // upper right dot
           g.fillOval(x+23, y+3, 9, 9);
       if (val == 6) // middle left dot
           g.fillOval(x+3, y+13, 9, 9);
       if (val % 2 == 1) // middle dot (for odd-numbered
val's)
           g.fillOval(x+13, y+13, 9, 9);
       if (val == 6) // middle right dot
           g.fillOval(x+23, y+13, 9, 9);
       if (val > 3)  // bottom left dot
           g.fillOval(x+3, y+23, 9, 9);
       if (val > 1)  // bottom right dot
           g.fillOval(x+23, y+23, 9,9);
   }


   /**
    * The draw() method just draws the two dice and draws
    * a two-pixel wide blue border around the canvas.
    */
```

```
    private void draw() {
        GraphicsContext g = canvas.getGraphicsContext2D();
        g.setFill(Color.rgb(200,200,255));
        g.fillRect(0,0,100,100);
        g.setStroke( Color.BLUE );
        g.strokeRect(1,1,98,98);
        drawDie(g, die1, 10, 10);
        drawDie(g, die2, 55, 55);
    }

} // end class RollDiceWithButton
```

Solution for Programming Exercise 6.5

Exercise 6.5:

In Exercise 3.8, you drew a checkerboard. For this exercise, write a program where the user can select a square by clicking on it. Highlight the selected square by drawing a colored border around it. When the program starts, 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 checkerboard is exactly 400 by 400 pixels, so that each square on the checkerboard is 50 by 50 pixels. Here is my checkerboard, with the square in row 3, column 3 selected, shown at reduced size:

Discussion

See the solution to Exercise 3.8 for a discussion of how to draw the checkerboard. For the new version, we need to add a `MousePressed` event handler to the *Canvas* on which the checkerboard is drawn. We also need to add instance variables to keep track of which square, if any, is selected. I use two integer variables `selectedRow` and `selectedCol`. When no square is selected, `selectedRow` is -1 (and I don't care what `selectedCol` is). When a square is selected, `selectedRow` is the number of the row that contains that square and `selectedCol` is the number of the column that contains the

selected square. Remember that rows and columns are numbered from 0 to 7. This makes some of the calculations easier than numbering them from 1 to 8.

After drawing the checkerboard, `draw()`, the method that draws the canvas, has to highlight the selected square, if there is one. I do this by drawing a cyan border around the inside of the selected square. This is the new code that is added to the checkerboard-drawing code, after the checkerboard has been drawn:

```
if (selectedRow >= 0) {
        // Since there is a selected square, draw a cyan
        // border around it.  (If selectedRow < 0, then
        // no square is selected and no border is drawn.)
    g.setStroke(Color.CYAN);
    g.setLineWidth(3);  // the border will be 3 pixels
wide
    y = selectedRow * 50;  // y-coord of top-left corner
of selected square
    x = selectedCol * 50;  // x-coord of top-left corner
of selected square
    g.strokeRect(x+1.5, y+1.5, 47, 47);
}
```

The border of the selected square is three pixels wide. To make the border lie entirely inside the square, we need to stroke a rectangle that is inset from the edges of the square by half of the line width, that is, by 1.5 pixels. This explains the parameters to `g.strokeRect()`. Remember that when a shape is stroked, it is the center of the pen that is dragged along the border of the shape.

Now, let's say that the user clicks at the point `(x,y)`. The problem is to determine which square on the checkerboard contains that point. The column number of the square is obtained by dividing the `x` coordinate by the width of the squares. The column number is an integer, while `x` is of type double. Since the squares are 50 pixels wide, the row number of the clicked square is `(int)(x/50)`. For values of `x` between 0 and 49, `x/50` is greater than or equal to 0 and less than 1, so taking its integer part gives a column number of 0, which is correct. For the next 50 pixels, from 50 to 99, `(int)(x/50)` is 1, which is the correct column number. For the next strip of pixels, from 100 to 149, the answer is 2. And so on. Similarly, `(int)(y/50)` gives the row

number of the square where the user clicked. (I often get rows and columns mixed up—remember that the x coordinate corresponds to columns and the y coordinate corresponds to rows.)

Once we know the row and column where the user clicked, we can compare them to `selectedRow` and `selectedCol`. If the values are the same, then the user clicked in a square that was already selected. We want to remove the highlighting. That can be done by setting `selectedRow = -1`, the value that indicates that no square is selected. Otherwise, the values of `selectedRow` and `selectedCol` are set to the row and column that the user clicked. In the end, `draw()` is called so that the change will be reflected in what is shown on the screen. The `draw()` method completely redraws the canvas, which is the easiest way to ensure that its appearance matches the current state of the program.

All this explains the reasoning behind the `mousePressed()` routine, which you can see below.

The Solution

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.layout.Pane;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.paint.Color;
import javafx.scene.input.MouseEvent;

/**
 *  This program draws a red-and-black checkerboard.
 *  It is assumed that the size of the canvas is 400
 *  by 400 pixels.  When the user clicks a square, that
 *  square is selected, unless it is already selected.
 *  When the user clicks the selected square, it is
 *  unselected.  If there is a selected square, it is
 *  highlighted with a cyan border.
 */
public class ClickableCheckerboard extends Application {

    public static void main(String[] args) {
```

```
        launch();
    }

    //-------------------------------------------------------
--------------

    private Canvas canvas;  // Where the checkerboard is
drawn

    private int selectedRow; // Row and column of selected
square.  If no
    private int selectedCol; //      square is selected,
selectedRow is -1.

    /**
     * Constructor.  Set selectedRow to -1 to indicate that
     * no square is selected.  And set the board object
     * to listen for mouse events on itself.
     */
    public void start(Stage stage) {

        selectedRow = -1;  // To start, no square is
selected!

        canvas = new Canvas(400,400);
        draw();

        canvas.setOnMousePressed(e -> mousePressed(e));

        Pane root = new Pane(canvas);
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setResizable(false);
        stage.setTitle("Click Me!");
        stage.show();
    }


    /**
     * Draw the checkerboard and highlight selected square,
if any.
     */
    private void draw() {

        int row;   // Row number, from 0 to 7
        int col;   // Column number, from 0 to 7
        int x,y;   // Top-left corner of square
```

```
        GraphicsContext g = canvas.getGraphicsContext2D();

        for ( row = 0;  row < 8;  row++ ) {

            for ( col = 0;  col < 8;  col++) {
                x = col * 50;
                y = row * 50;
                if ( (row % 2) == (col % 2) )
                    g.setFill(Color.RED);
                else
                    g.setFill(Color.BLACK);
                g.fillRect(x, y, 50, 50);
            }

        } // end for row

        if (selectedRow >= 0) {
                // Since there is a selected square, draw a
cyan
                // border around it.  (If selectedRow < 0,
then
                // no square is selected and no border is
drawn.)
            g.setStroke(Color.CYAN);
            g.setLineWidth(3);
            y = selectedRow * 50;
            x = selectedCol * 50;
            g.strokeRect(x+1.5, y+1.5, 47, 47);
        }

    }  // end paint()


    /**
     * When the user clicks on the canvas, figure out which
     * row and column the click was in and change the
     * selected square accordingly.
     */
    private void mousePressed(MouseEvent evt) {

        int col = (int)(evt.getX() / 50);   // Column where
user clicked.
        int row = (int)(evt.getY() / 50);   // Row where
user clicked.

        if (selectedRow == row && selectedCol == col) {
                // User clicked on the currently selected
square.
```

```
                // Turn off the selection by setting
selectedRow to -1.
            selectedRow = -1;
        }
        else {
                // Change the selection to the square the
user clicked on.
            selectedRow = row;
            selectedCol = col;
        }
        draw();

    }  // end mousePressed()

} // end ClickableCheckerboard
```

Solution for Programming Exercise 6.6

Exercise 6.6:

For this exercise, you should modify the SubKiller game from Subsection 6.3.6. You can start with the existing source code, from the file SubKiller.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 canvas 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.

Discussion

You can do this exercise by adding as few as seven lines to the original version, SubKiller.java (plus changing the name of the class, if you want to do that). I used two lines to declare instance variables named `hits` and `misses`. These variables have to be updated whenever the depth charge hits the sub or falls off the bottom of the canvas. These events are already detected by the canvas, in the `updateForNextFrame()` method of the *Bomb* class. At the point where this method detects that the depth charge has hit the sub, I add the command "`hits++;`" to chalk up another hit for the user. At the point in where it is determined that the `y`-coordinate of the depth charge has exceeded the height of the canvas, I add the command "`misses++;`" to record the fact that the sub has escaped destruction this time.

The only other thing to do is to display the number of hits and misses at the top of the canvas. This is part of drawing the picture, so it is done in the

`draw()` method. The information is output with two `drawString` commands. We need one more command to set the color for the strings:

```
g.setFill(Color.BLACK);
g.fillText("Number of hits:   " + hits, 15, 24);
g.fillText("Number of misses: " + misses, 15, 45);
```

(Using the coordinates in these statements, the messages about hits and misses were drawn over the "(ANIMATION PAUSED)" message, so I also moved the "(ANIMATION PAUSED)" message from x-coordinate 30 to x-coordinate 300 to make room for the new messages.)

I made one further non-essential change when I decided that I wanted the output to be displayed in a larger font. I create a new font, store it in an instance variable named `infoFont`, and use the command "`g.setFont(infoFont);`" before drawing the strings.

The source code is shown below. Changes from the original version are shown in red italic.

The Solution

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.stage.Stage;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.input.KeyCode;
import javafx.animation.AnimationTimer;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;

/**
 * This program implements a simple arcade game in which
the user tries to blow
 * up a "submarine" (a black oval) by dropping "depth
charges" (a red disk) from
 * a "boat" (a blue roundrect).  The user moves the boat
with the left- and
 * right-arrow keys and drops the depth charge with the
```

```
down-arrow key.
 * The sub moves left and right erratically along the
bottom of the canvas.
 * The number of hits and the number of misses are shown at
the top of the window.
 */
public class SubKillerWithScore extends Application {

    public static void main(String[] args) {
        launch();
    }

    //-------------------------------------------------------
-------------------

    private AnimationTimer timer; // AnimationTimer that
drives the animation.

    private final int width = 640, height = 480; // The
size of the canvas

    private Boat boat;          // The boat, bomb, and sub
objects are defined
    private Bomb bomb;          //    by nested classes
Boat, Bomb, and Submarine,
    private Submarine sub;      //    which are defined
later in this class.

    private int hits;           // The number of times the
user has hit the sub.
    private int misses;         // The number of times the
user has missed the sub.

    private Font infoFont = Font.font(null,
FontWeight.BOLD, 16);
                            // A font for displaying the
numbers of hits and misses.

    private Canvas canvas;      // The canvas where
everything is drawn.

    private Stage stage;        // This program's window.


    /**
     * Start method creates the window content and
configures event listening.
```

```
     */
    public void start(Stage stage) {

        /* Create objects. */

        boat = new Boat();
        sub = new Submarine();
        bomb = new Bomb();

        canvas = new Canvas(width,height);

        this.stage = stage;

        /* Set up the GUI */

        Pane root = new Pane(canvas);
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Sub Killer -- Use arrow keys to
play!");
        stage.setResizable(false);

        /* Configure event listeners and animation. */

        scene.setOnKeyPressed( evt -> {
                // The key listener responds to keyPressed
events on the canvas. Only
                // the left-, right-, and down-arrow keys
have any effect.  The left- and
                // right-arrow keys move the boat while
down-arrow releases the bomb.
            KeyCode code = evt.getCode();  // Which key was
pressed?
            if (code == KeyCode.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 == KeyCode.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 == KeyCode.DOWN) {
```

```
                    // Start the bomb falling, if it is not
already falling.
                if ( bomb.isFalling == false )
                    bomb.isFalling = true;
            }
        } );

        stage.focusedProperty().addListener(
(obj,oldVal,newVal) -> {
                // This listener turns the animation off
when this program's
                // window does not have the input focus.
            if (newVal) { // The window has gained focus.
                timer.start();
            }
            else {  // The window has lost focus.
                timer.stop();
            }
            draw(); // Appearance changes depending on
focus.
        });

        timer = new AnimationTimer( ) {
               // The handle method is called once per
frame while the
               // animation is running.  There should be
about 60
               // frames per second.
            long previousFrameTime;
            public void handle(long time) {
                if (time - previousFrameTime > 0.99e9/60) {
                        // This program is designed to run
at 60 frames per
                        // second.  Since handle() can be
called more frequently
                        // than that, we do an update-and-
draw only if the
                        // time since the previous update-
and-draw is greater
                        // than 0.99/60 seconds.  Note that
time is measured in
                        // nanoseconds, where one second
equals 1e9 nanoseconds.
                    boat.updateForNewFrame();
                    bomb.updateForNewFrame();
                    sub.updateForNewFrame();
                    draw();
                    previousFrameTime = time;
```

```
                }
            }
        };

        /* Show the window. */

        stage.show();
        timer.start(); // (Start animation -- probably
redundant, since
                        //  focus listener should be called
when window opens.)
    } // end start()


    /**
     * The draw() method draws the current state of the
game.  It
     * draws a gray or cyan border around the canvas to
indicate whether or not
     * the window has the input focus.  It draws the boat,
sub, and bomb by
     * calling their respective draw() methods.  This
method is called in
     * each frame and when the window gains or loses focus.
     */
    public void draw() {

        GraphicsContext g = canvas.getGraphicsContext2D();
        g.setFont(infoFont);

        g.setFill(Color.rgb(0,200,0)); // Fill with green,
erasing previous frame.
        g.fillRect(0,0,width,height);

        if (stage.isFocused()) {
                // draw a CYAN border on the window when
the window has focus.
            g.setStroke(Color.CYAN);
        }
        else {
                // Draw gray border and a message when
window does not have focus.
            g.setFill(Color.BLACK);
            g.fillText("(ANIMATION PAUSED)", 300, 30); //
(x-coord was changed)
            g.setStroke(Color.DARKGRAY);
        }
```

```
        g.setLineWidth(3);
        g.strokeRect(1.5,1.5,width-3,height-3);  // Draw a
3-pixel border.

        boat.draw(g);  // The three objects draw
themselves.
        sub.draw(g);
        bomb.draw(g);

        g.setFill(Color.BLACK);
        g.fillText("Number of hits:   " + hits, 15, 24);
        g.fillText("Number of misses: " + misses, 15, 45);

    } // end draw()


    /**
     * This nested class defines the boat.
     */
    private class Boat {
        int centerX, centerY;  // Current position of the
center of the boat.
        Boat() { // Constructor centers the boat
horizontally, 80 pixels from top.
            centerX = width/2;
            centerY = 80;
        }
        void updateForNewFrame() { // Makes sure boat has
not moved off screen.
            if (centerX < 0)
                centerX = 0;
            else if (centerX > width)
                centerX = width;
        }
        void draw(GraphicsContext g) {  // Draws the boat
at its current location.
            g.setFill(Color.BLUE);
            g.fillRoundRect(centerX - 40, centerY - 20, 80,
40, 20, 20);
        }
    } // end nested class Boat


    /**
     * This nested class defines the bomb.
     */
    private class Bomb {
```

```
        int centerX, centerY; // Current position of the
center of the bomb.
        boolean isFalling;    // If true, the bomb is
falling; if false, it
                              // is attached to the boat.
        Bomb() { // Constructor creates a bomb that is
initially attached to boat.
            isFalling = false;
        }
        void updateForNewFrame() {  // If bomb is falling,
take appropriate action.
            if (isFalling) {
                if (centerY > height) {
                        // Bomb has missed the submarine.
It is returned to its
                        // initial state, with isFalling
equal to false.
                    isFalling = false;
                    misses++; // The number of misses goes
up by one.
                }
                else if (Math.abs(centerX - sub.centerX) <=
36 &&
                        Math.abs(centerY - sub.centerY) <=
21) {
                        // Bomb has hit the submarine.  The
submarine
                        // enters the "isExploding" state.
                    sub.isExploding = true;
                    sub.explosionFrameNumber = 1;
                    isFalling = false;  // Bomb reappears
on the boat.
                    hits++;  // The number of hits goes up
by one.
                }
                else {
                        // If the bomb has not fallen off
the canvas or hit the
                        // sub, then it is moved down 6
pixels.
                    centerY += 6;
                }
            }
        }
        void draw(GraphicsContext g) { // Draw the bomb.
            if ( ! isFalling ) {  // If not falling, set
centerX and centerY
```

```
                                    // to show the bomb on
the bottom of the boat.
                centerX = boat.centerX;
                centerY = boat.centerY + 23;
            }
            g.setFill(Color.RED);
            g.fillOval(centerX - 8, centerY - 8, 16, 16);
        }
    } // end nested class Bomb


    /**
     * This nested class defines the sub.
     */
    private class Submarine {
        int centerX, centerY; // Current position of the
center of the sub.
        boolean isMovingLeft; // Tells whether the sub is
moving left or right
        boolean isExploding;  // Set to true when the sub
is hit by the bomb.
        int explosionFrameNumber;  // If the sub is
exploding, this is the number
                                    //   of frames since the
explosion started.
        Submarine() {  // Create the sub at a random x-
coordinate, 40 pixels from bottom.
            centerX = (int)(width*Math.random());
            centerY = height - 40;
            isExploding = false;
            isMovingLeft = (Math.random() < 0.5);
        }
        void updateForNewFrame() { // Move sub or increase
explosionFrameNumber.
            if (isExploding) {
                    // If the sub is exploding, add 1 to
explosionFrameNumber.
                    // When the number reaches 25, the
explosion ends and the
                    // sub reappears in a random position.
                explosionFrameNumber++;
                if (explosionFrameNumber == 25) {
                    centerX = (int)(width*Math.random());
                    centerY = height - 40;
                    isExploding = false;
                    isMovingLeft = (Math.random() < 0.5);
                }
            }
```

```
            else { // Move the sub.
                if (Math.random() < 0.02) {
                        // In one frame out of every 50, on
average, the sub
                        // reverses its direction of
motion.
                    isMovingLeft = ! isMovingLeft;
                }
                if (isMovingLeft) {
                        // Move the sub 3 pixels to the
left.  If it moves off
                        // the left edge of the canvas,
move it back to the left
                        // edge and start it moving to the
right.
                    centerX -= 3;
                    if (centerX <= 0) {
                        centerX = 0;
                        isMovingLeft = false;
                    }
                }
                else {
                        // Move the sub 3 pixels to the
right.  If it moves off
                        // the right edge of the canvas,
move it back to the right
                        // edge and start it moving to the
left.
                    centerX += 3;
                    if (centerX > width) {
                        centerX = width;
                        isMovingLeft = true;
                    }
                }
            }
        }
        void draw(GraphicsContext g) {  // Draw sub and, if
it is exploding, the explosion.
            g.setFill(Color.BLACK);
            g.fillOval(centerX - 30, centerY - 15, 60, 30);
            if (isExploding) {
                    // Draw an "explosion" that grows in
size as the number of
                    // frames since the start of the
explosion increases.
                g.setFill(Color.YELLOW);
                g.fillOval(centerX -
3*explosionFrameNumber,
```

```
                        centerY - 1.5*explosionFrameNumber,
                        6*explosionFrameNumber,
                        3*explosionFrameNumber);
                g.setFill(Color.RED);
                g.fillOval(centerX -
1.5*explosionFrameNumber,
                        centerY - explosionFrameNumber/2,
                        3*explosionFrameNumber,
                        explosionFrameNumber);
            }
        }
    } // end nested class Submarine


} // end class SubKillerWithScore
```

Solution for Programming Exercise 6.7

Exercise 6.7:

Exercise 5.2 involved a class, StatCalc.java, that could compute some statistics of a set of numbers. Write a GUI program that uses the *StatCalc* class to compute and display statistics of numbers entered by the user. The program will have an instance variable of type *StatCalc* that does the computations. The program should include a *TextField* 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 *TextField* and then either clicking an "Enter" button or pressing the Return (or Enter) key. There should be a "Clear" button that clears out all the data. This means creating a new *StatCalc* object and changing the text that is displayed on the labels. (See the discussion of "default buttons" at the end of Subsection 6.4.2 for one way of implementing a response the Return key.) Here is a picture of my solution to this problem:

Getting the interface to look just like I wanted it was the hardest part. In the end, I used *TilePanes* (Subsection 6.5.4) for the layout.

Discussion

In my solution, I used four labels to display results from the *StatCalc*, and another label at the top of the window to display a message to the user. Aside from these labels, one row in the layout holds the other three components: a *TextField* and two *Buttons*. I want to spend some time talking about how I designed the GUI for the program. My first idea for layout was to use an *HBox* to hold the text field and buttons, and a *VBox* to line up the *HBox* and the five labels vertically. As I often do, I first wrote the program without paying too much attention to appearance. The result was not very attractive:

Note in particular that there is not enough room in the bottom display labels to display the numbers. When this happens, a *Label* will automatically modify the displayed string to use an ellipsis (...) to indicate that not all of the text is shown. (Buttons will do the same thing.) When the labels were constructed, the text in the constructor did not include the numbers, so the preferred size of the labels was not large enough to allow enough space when the numbers were added later. This is often an issue when using labels and buttons whose text will change while the program is running.

I could have worked to improve the appearance, but I decided that it would be easier to do if I used *TilePanes* for layout. My final program uses a *TilePane* to hold the textfield and buttons; the preferred number of columns for that *TilePane* is set to 3. And the program uses a *TilePane* with one column for the overall layout.

Labels and buttons in a *TilePane* will not automatically expand to fill their tiles, because of their default maximum size. To allow them to expand, we have to increase their maximum sizes. This can be done by calling, for example, `label.setMaxSize(1000,1000)`.

I also reduced the number of preferred columns in the *TextField*, since the default forced the preferred size of the scene to be larger than I liked. With

all of these changes, the interface looked much nicer:

This would have been satisfactory, but I wanted something a little more dramatic. I decided to add lines between the labels. I could have done that by adding black borders to the labels, but I decided to use a different approach. I set the horizontal and vertical gaps on the *TilePanes* to be 3 pixels. The background will show through the gaps, so I got my black lines by setting the background color of the root *TilePane* to be black, instead of the usual light gray. Unfortunately, the default background for a label is transparent, which meant that all the labels were filled with the black background of the root *TilePane*, hiding the text, which was also black! To fix that, I set the text color for the top label to be white, and I set the background color for the other four labels to be white. This finally gave me the look that I wanted. I should emphasize that the approach that I used is not by any means the only one that would have worked.

Turning to the actual programming, consider the labels that display the results from the *StatCalc*. Since there are four labels to create, I wrote a method to create and configure them:

```
/**
 * A utility routine for creating the labels that are used
 * for display.  This routine is used in the start()
method.
 */
private Label makeLabel(String text) {
   Label label = new Label(text);
   label.setMaxSize(1000,1000);
   label.setStyle("-fx-background-color:white; " +
                        "-fx-font-family: monospace; -fx-
font-weight: bold");
```

```
    return label;
}
```

Note that the maximum width and height of the label have been made quite large. CSS styling (Subsection 6.2.5) is used to set the background color to be white and to use a bold monospace font. (A monospace font is used since every character in a monospace font has the same width, making it easy to line up the numbers vertically in the labels. JavaFX seems to have no way to ensure that a font will be monospace, except to use CSS.)

Then in the `start()` method, the labels can be created with four statements, instead of 12:

```
countLabel =   makeLabel(" Number of Entries:  0");
sumLabel =     makeLabel(" Sum:                0.0");
meanLabel =    makeLabel(" Average:
undefined");
standevLabel = makeLabel(" Standard Deviation:
undefined");
```

Utility routines like `makeLabel()` are very commonly used when there are a lot of similar components to create. Note that when the labels are first created, the text on the labels is appropriate for a dataset that contains zero elements. In particular, if there is no data, the average and standard deviation are undefined.

An `ActionEvent` handler is installed on each button. The handler calls an appropriate method that is defined elsewhere in the class. The "Enter" button is made into the default button for the program by calling `enterButton.setDefaultButton(true)`, as suggested in the instructions for the exercise. Note in the screenshot from the program that this also changes the appearance of the "Enter" button. Here is the code that creates and configures the buttons:

```
enterButton = new Button("Enter");
enterButton.setOnAction( e -> doEnter() );
enterButton.setMaxSize(1000,1000);
enterButton.setDefaultButton(true);

clearButton = new Button("Clear");
```

```
clearButton.setOnAction( e -> doClear() );
clearButton.setMaxSize(1000,1000);
```

In the `doClear()` method, the response to the "Clear" button is to create a new *StatCalc* object and to reset the display labels to reflect the fact that there is no data in the dataset. The labels are changed by another method, `showData()`:

```
private void doClear() {
    stats = new StatCalc();
    numberInput.setText("");
    showData();
}
```

Here, `stats` is a global variable of type `StatCalc` that is used to do the calculations in the program. It's important to understand the effect of the command "`stats = new StatCalc();`". The program will continue to use the same *StatCalc* **variable**, `stats`. However, now the variable refers to a new *StatCalc* **object**. The new object does not yet have any data in its dataset. When `showData()` is called, it is working with a `StatCalc` object to which no data has been added. The next time the user enters a number, the dataset will get its first value. Always keep in mind the difference between variables and objects. Also, keep in mind that you have to think in terms of changing the state of the program in response to events. I change the program's state by starting to use a new *StatCalc* object, and I call `showData()` to change the display labels to keep them consistent with the new state.

When the user clicks the "Enter" button or presses Return, another method, `doEnter()` is called. That method has to get the user's input, add it to the *StatCalc* object, and then call `showData()` to change the display labels to show the new values. The code for getting the user's number from the input box includes a check to make sure that the user's input is a legal number. If the input is not legal, then the method shows an error message in the *Label* at the top of the window, which is represented by a global variable named `message`:

```
private void doEnter() {
    double num;  // The user's number.
    try {
```

```
        num = Double.parseDouble(numberInput.getText());
    }
    catch (NumberFormatException e) {
            // The user's entry is not a legal number.
            // Put an error message in the message label
            // and return without entering a number.
        message.setText("\"" + numberInput.getText() + "\"
is not a legal number.");
        numberInput.selectAll();
        numberInput.requestFocus();
        return;
    }
    stats.enter(num);
    showData();
}
```

When an error is detected, the method returns immediately, since there is no data to process. The commands "`numberInput.selectAll();`" and "`numberInput.requestFocus();`" are there as a convenience for the user. The first command selects all the text in the textfield. The second command gives the input focus to the textfield. That way, the user can just start typing the next number, without having to click on the input box or erase the content of the box. (Since the contents of the box are selected, they will disappear automatically when the user starts typing, to be replaced with the new input. A surprising number of people have never learned that text selections work this way.)

Once we have the user's number, the command "`stats.enter(num);`" adds the number `num` into the dataset. In the `showData()` method, the statistics about the data set can be obtained by calling the functions `stats.getCount()`, `stats.getSum()`, `stats.getMean()`, and `stats.getStandardDeviation()`. (This information can be found by reading the source code for the *StatCalc* class.) These values are used on the labels that display the statistics. For example,

```
countLabel.setText(" Number of Entries:  " +
stats.getCount());
```

Returning to the topic of variables versus objects, a common novice mistake would be to try to change the text that is displayed on the label by saying

```
countLabel = new Label(" Number of Entries:  " +
stats.getCount());  // WRONG
```

since that type of command is what was used to set the text in the first place. But this statement has no effect on what is displayed on the screen. The reason why this doesn't work is instructive: The assignment command creates a new *Label* and sets `countLabel` to refer to that new object. The new label does indeed have the desired text. But the new label has nothing to do with what's on the screen! The original label was created in the constructor and added to the root tilepane. It is the **original** label object that is in the scene graph, so the original label continues to appear on the screen even if the instance variable `countLabel` now refers to a different label object. The correct command, using `countLabel.setText()`, **modifies the original label object**, which is what is shown on the screen.

As a point of interest, I will mention another technique. You might want to limit the number of decimal places that are displayed in the statistics. You know how to do this in output to the command line, using `System.out.printf`, but we are not outputting the data here, just putting it into a String. The *String* class has a `static` method `String.format()` that solves this problem. `String.format(fmtString,val,val,val,...)` works similarly to `System.out.printf(fmtString,val,val,val,...)`, but instead of producing output, `String.format` just creates a string and returns it. To set the text of the label that displays the average, for example, you Might say:

```
meanLabel.setText( String.format(" Average:
%1.8g", stats.getMean()) );
```

I don't do this in my solution, but `String.format()` can be very useful

The Solution

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.TilePane;
import javafx.scene.control.Button;
```

```
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;

/**
 * In this program, the user enters numbers in a text field

box.
 * After entering each number, the user presses return (or
clicks
 * on a button).  Some statistics are displayed about all
the
 * numbers that the user has entered.
 */
public class StatCalcGUI extends Application {

    public static void main(String[] args) {
        launch();
    }

    //-----------------------------------------------------
----------------

    private Label countLabel;    // A label for displaying
the number of numbers.
    private Label sumLabel;      // A label for displaying
the sum of the numbers.
    private Label meanLabel;     // A label for displaying
the average.
    private Label standevLabel;  // A label for displaying
the standard deviation.

    private Label message;  // A message at the top of the
program.  It will
                            //   show an error message if
the user's input is
                            //   not a legal number.
Otherwise, it just tells
                            //   the user to enter a number
and press return.

    private Button enterButton;   // A button the user can
press to enter a number.
                                  //    This is an
alternative to pressing return.

    private Button clearButton;   // A button that clears
all the data that the
```

```
all the data that the
                                    //    user has entered.

    private TextField numberInput;  // The input box where
the user enters numbers.

    private StatCalc stats;  // An object that keeps track
of the statistics
                             //   for all the numbers that
have been entered.


    /**
     * Set up the GUI and event handling.
     */
    public void start(Stage stage) {

       stats = new StatCalc();

       numberInput = new TextField();
       numberInput.setPrefColumnCount(8);  // Makes the
text box smaller than the default.

       enterButton = new Button("Enter");
       enterButton.setOnAction( e -> doEnter() );
       enterButton.setMaxSize(1000,1000);
       enterButton.setDefaultButton(true); // Pressing
return will be equivalent to clicking this button.
       clearButton = new Button("Clear");
       clearButton.setOnAction( e -> doClear() );
       clearButton.setMaxSize(1000,1000);

       countLabel =   makeLabel(" Number of Entries:  0");
       sumLabel =     makeLabel(" Sum:
0.0");
       meanLabel =    makeLabel(" Average:
undefined");
       standevLabel = makeLabel(" Standard Deviation:
undefined");

       message = new Label("Enter a number, press
return:");
       message.setFont(Font.font(16));
       message.setTextFill(Color.WHITE);

       TilePane inputPanel = new
TilePane(3,3,numberInput,enterButton,clearButton);
       inputPanel.setPrefColumns(3);
       TilePane root = new TilePane(3, 3, message
```

```
        TilePane root = new TilePane(5, 5, message,
inputPanel, countLabel,
                                                    sumLabel,
meanLabel, standevLabel);
        root.setPrefColumns(1);
        root.setStyle("-fx-border-color:black; " +
                              "-fx-border-width:3; -fx-
background-color:black");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Simple Calc GUI");
        stage.setResizable(false);
        stage.show();

    } // end start()


    /**
     * A utility routine for creating the labels that are
used
     * for display.  This routine is used in the start()
method.
     * @param text The text to show on the label.
     */
    private Label makeLabel(String text) {
        Label label = new Label(text);
        label.setMaxSize(1000,1000);
        label.setStyle("-fx-background-color:white; " +
                                 "-fx-font-family: monospace;
-fx-font-weight: bold");
        return label;
    }


    /**
     * Clear all data, restoring the program to its
original state.
     * This method is called when the user clicks the Clear
button.
     */
    private void doClear() {
        stats = new StatCalc();
        numberInput.setText("");
        showData();
    }


    /**
```

```
    /**
     * Respond when the clicks the Enter button by getting
a number from
     * the text input box, adding it to the StatCalc and
updating the
     * four display labels.  It is possible that an error
will occur,
     * in which case an error message is put into the label
at the top
     * of the window.  (Because the Enter button has been
set to be the
     * default button for the program, this method is also
invoked when
     * the user presses return.)
     */
    private void doEnter() {
        double num;  // The user's number.
        try {
            num =
Double.parseDouble(numberInput.getText());
        }
        catch (NumberFormatException e) {
                // The user's entry is not a legal number.
                // Put an error message in the message
label
                // and return without entering a number.
            message.setText("\"" + numberInput.getText() +
"\" is not a legal number.");
            numberInput.selectAll();
            numberInput.requestFocus();
            return;
        }
        stats.enter(num);
        showData();
    }


    /**
     *  Show the data from the StatCalc in the four output
labels.
     */
    private void showData() {
        countLabel.setText(" Number of Entries:  " +
stats.getCount());
        sumLabel.setText(" Sum:                " +
stats.getSum());
        if (stats.getCount() == 0) {
               // Don't show any values for mean and
standard deviation if
```

```
standard deviation if
                // no numbers have been added to the data.
            meanLabel.setText(" Average:
undefined");
            standevLabel.setText(" Standard Deviation:
undefined");
        }
        else {
            meanLabel.setText(" Average:            " +
stats.getMean());
            standevLabel.setText(" Standard Deviation: " +
stats.getStandardDeviation());
        }

        /* Set the message label back to its normal text,
in case it has
          been showing an error message.  For the user's
convenience,
          select the text in the TextField and give the
input focus
          to the text field.  That way the user can just
start typing
          the next number. */

        message.setText("Enter a number, click Enter:");
        numberInput.selectAll();
        numberInput.requestFocus();
    }


}  // end StatsCalcGUI
```

Solution for Programming Exercise 6.8

Exercise 6.8:

Write a program that has a *TextArea* where the user can enter some text. The program should have a button such that when the user clicks on the button, the program will 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*. Recall that if `textInput` is a *TextArea*, then you can get the contents of the *TextArea* 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. Here is a picture of my solution:

Discussion

The window contains five components. There are several ways to lay them out. Since the *TextArea* is much larger than the other components, my program uses a *VBox* as the `root` of the scene graph, which allows each component to have its preferred height.

To get a dark blue border around the edges, I use CSS to add a border to the *VBox*. To get blue lines between the components, I set the spacing between components in the *VBox* to be 4, and I make its background dark blue. I set the background color for the labels to be white; otherwise, the usual transparent background for the labels would let the blue background of the *VBox* show through. I increased the maximum width of the labels to make them fill the entire width of the *VBox*. (Otherwise, the blue background would show through in the space left empty by the labels.) As for the button, I did not want to increase its maximum width to fill the available space, but without that, the *VBox* placed it on the left edge of the available space. I wanted it to be centered. The only way I could think of to do that was to put the button inside another pane, and add that pane to the *VBox*. That pane will fill the available space. I used a *BorderPane* with the button as its center component; the parameter to the *BorderPane*constructor becomes the center component in the layout, and a *BorderPane* centers its center component within the available space. You can see how it's is all done in this code from the `start()` method:

```
textInput = new TextArea();
    textInput.setPrefRowCount(15);
    textInput.setPrefColumnCount(30);

    /* Create the button and a listener to listen for
         clicks on the button. */

    Button countButton = new Button("Process the Text");
    countButton.setOnAction( e -> processInput() );

    /* Create each of the labels, and set their
properties. */

    String style =
         "-fx-padding: 5px; -fx-font: bold 14pt serif; -
```

```
fx-background-color: white";

    lineCountLabel = new Label("  Number of lines:");
    lineCountLabel.setStyle(style);
    lineCountLabel.setMaxWidth(1000);

    wordCountLabel = new Label("  Number of words:");
    wordCountLabel.setStyle(style);
    wordCountLabel.setMaxWidth(1000);

    charCountLabel = new Label("  Number of chars:");
    charCountLabel.setStyle(style);
    charCountLabel.setMaxWidth(1000);

    /* Use a VBox as the root component. */

    VBox root = new VBox( 4, textInput, new
BorderPane(countButton),
                                lineCountLabel,
wordCountLabel, charCountLabel );
    root.setStyle(
        "-fx-background-color: #009; -fx-border-color:
#009; -fx-border-width:3px" );
```

The first parameter to the *VBox* constructor is the spacing between components, and the remaining parameter are added as child nodes of the *VBox*. Note that `countButton` is wrapped in a pane which is then added to the *VBox*.

The `ActionEvent` handler for the button calls the method `processInput()`. That method does all the work of the program: It gets the text from the `TextArea`, does the counting, and sets the labels. The only interesting part is counting the words. Back in Exercise 3.4, words such as "can't", that contain an apostrophe, were counted as two words. This time around, let's handle this special case. Two letters with an apostrophe between them should be counted as part of the same word. The algorithm for counting words is still

```
wordCt = 0
for each character in the string:
   if the character is the first character of a word:
      Add 1 to wordCt
```

but testing whether a given character is the first character in a word has gotten more complicated. To make the test easier, I use a boolean variable, `startOfWord`. The value of this variable is set to true if the character is the start of a word and to false if not. That is, the algorithm becomes:

```
wordCt = 0
for each character in the string:
   Let startOfWord be true if at start of word, false
otherwise
   if startOfWord is true:
      Add 1 to wordCt
```

The use of a "flag variable" like `startOfWord` can simplify the calculation of a complicated boolean condition. The value is computed as a series of tests:

```
boolean startOfWord;  // Is character i the start of a
word?
if ( Character.isLetter(text.charAt(i)) == false )
   startOfWord = false;  // No.  It's not a letter.
else if (i == 0)
   startOfWord = true;   // Yes.  It's a letter at start
of text.
else if ( Character.isLetter(text.charAt(i-1)) )
   startOfWord = false;  // No.  It's a letter preceded by
a letter.
else if ( text.charAt(i-1) == '\'' && i > 1
                     && Character.isLetter(text.charAt(i-
2)) )
   startOfWord = false;  // No.  It's a continuation of a
word
                         //      after an apostrophe.
else
   startOfWord = true;   // Yes.  It's a letter preceded
by
                         //       a non-letter.
```

The first test checks whether the character in position `i` is a letter. If it is not, then we know that it can't be the start of a word, so `startOfWord` is false. If it is a letter, it might be the start of a word, so we go on to make additional tests. Note that if we get to the other tests at all, we already know that the character in position `i` is a letter. And so on. This style of "cascading tests" is very useful. In each test, we already have all the information from the

previous tests. Note that the cascade effect works only with "`else if`". Using "`if`" in place of "`else if`" in the preceding code would not give the right answer. (You should be sure to understand why this is so.) You should also note why the test "`if (i == 0)`"" has to be made before the test `if ( Character.isLetter(text.charAt(i-1)) )`"—that's because `text.charAt(i-1)` gives an index-out-of-bounds exception if `i` is zero.

The Solution

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.VBox;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextArea;


/**
 * In this program, the user types some text in a TextArea
and presses
 * a button.  The program computes and displays the number
of lines
 * in the text, the number of words in the text, and the
number of
 * characters in the text.  A word is defined to be a
sequence of
 * letters, except that an apostrophe with a letter on each
side
 * of it is considered to be a letter.  (Thus "can't" is
one word,
 * not two.)
 */
public class TextCounter extends Application {

    public static void main(String[] args) {
        launch(args);
    }

    //-------------------------------------------------------
----------------

    private TextArea textInput;     // For the user's input
```

text.

```
    private Label lineCountLabel;   // For displaying the
number of lines.
    private Label wordCountLabel;   // For displaying the
number of words.
    private Label charCountLabel;   // For displaying the
number of chars.


    /**
     * The constructor creates components and lays out the
window.
     */
    public void start(Stage stage) {

        textInput = new TextArea();
        textInput.setPrefRowCount(15);
        textInput.setPrefColumnCount(30);

        /* Create the button and a listener to listen for
             clicks on the button. */

        Button countButton = new Button("Process the
Text");
        countButton.setOnAction( e -> processInput() );

        /* Create each of the labels, and set their
properties. */

        String style = "-fx-padding: 5px; -fx-font: bold
14pt serif; -fx-background-color: white";

        lineCountLabel = new Label("  Number of lines:");
        lineCountLabel.setStyle(style);
        lineCountLabel.setMaxWidth(1000);

        wordCountLabel = new Label("  Number of words:");
        wordCountLabel.setStyle(style);
        wordCountLabel.setMaxWidth(1000);

        charCountLabel = new Label("  Number of chars:");
        charCountLabel.setStyle(style);
        charCountLabel.setMaxWidth(1000);

        /* Use a VBox as the root component. */

        VBox root = new VBox( 4, textInput, new
```

```
BorderPane(countButton),
                                    lineCountLabel,
wordCountLabel, charCountLabel );
        root.setStyle("-fx-background-color: #009; -fx-
border-color: #009; -fx-border-width:3px");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Line/Word/Char Counter");
        stage.setResizable(false);
        stage.show();

    } // end constructor


    /**
     * This will be called when the user clicks the
"Process the Text" button.
     * It gets the text from the text area, counts the
number of chars, words,
     * and lines that it contains, and sets the labels to
display the results.
     */
    public void processInput() {

        String text;  // The user's input from the text
area.

        int charCt, wordCt, lineCt;  // Char, word, and
line counts.

        text = textInput.getText();

        charCt = text.length();  // The number of
characters in the
                                 //    text is just its
length.

        /* Compute the wordCt by counting the number of
characters
              in the text that lie at the beginning of a
word.  The
              beginning of a word is a letter such that the
preceding
              character is not a letter.  This is
complicated by two
              things:  If the letter is the first character
in the
```

```
              text, then it is the beginning of a word.  If
the letter
              is preceded by an apostrophe, and the
apostrophe is
              preceded by a letter, than its not the first
character
              in a word.
         */

        wordCt = 0;
        for (int i = 0; i < charCt; i++) {
            boolean startOfWord;  // Is character i the
start of a word?
            if ( Character.isLetter(text.charAt(i)) ==
false )
                startOfWord = false;  // No.  It's not a
letter.
            else if (i == 0)
                startOfWord = true;   // Yes.  It's a
letter at start of text.
            else if ( Character.isLetter(text.charAt(i-1))
)
                startOfWord = false;  // No.  It's a letter
preceded by a letter.
            else if ( text.charAt(i-1) == '\'' && i > 1
                    && Character.isLetter(text.charAt(i-2))
)
                startOfWord = false;  // No.  It's a
continuation of a word
                                      //      after an
apostrophe.
            else
                startOfWord = true;   // Yes.  It's a
letter preceded by
                                      //       a non-
letter.
            if (startOfWord)
                wordCt++;
        }

        /* The number of lines is just one plus the number
of times the
              end of line character, '\n', occurs in the
text. */

        lineCt = 1;
        for (int i = 0; i < charCt; i++) {
            if (text.charAt(i) == '\n')
```

```
                lineCt++;
        }

        /* Set the labels to display the data. */

        lineCountLabel.setText("  Number of lines:  " +
lineCt);
        wordCountLabel.setText("  Number of words:  " +
wordCt);
        charCountLabel.setText("  Number of chars:  " +
charCt);

    }  // end processInput()


} // end class TextCounter
```

Solution for Programming Exercise 6.9

Exercise 6.9:

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. Subsection 6.2.4 has a list of shape-drawing methods in a *GraphicsContext*. Among them are methods for stroking and for filling polygons: `g.strokePolygon(xcoords,ycoords,n)` and `g.fillPolygon(xcoords,ycoords,n)`, For these commands, the coordinates of the vertices of the polygon are stored in arrays `xcoords` and `ycoords` of type `double[]`, and the number of vertices of the polygon is given by the third parameter, `n`. Note that it is OK for the sides of a polygon to cross each other, but the interior of a polygon with self-intersections might not be exactly what you expect.

Write a program that lets the user draw polygons. As the user clicks a sequence of points in a *Canvas*, count the points and store their x- and y-coordinates in two arrays. These points will be the vertices of the polygon. As the user is creating the polygon, you should just connect all the points with line segments. When the user clicks near the starting point, draw the complete polygon. Draw it with a red interior and a black border. Once the user has completed a polygon, the next click should clear the data and start a new polygon from scratch.

Here is a picture of my solution after the user has drawn a fairly complex polygon:

Discussion

This is an exercise in using arrays, but it is also an exercise in using instance variables to record the state of the program. When the user clicks the canvas, my program completely redraws the canvas. (This is not really necessary, but it's often the easiest way to organize a drawing program, since it's the easiest way to ensure that the picture always agrees perfectly with the state of the program.) The `draw()` method that draws the canvas needs enough state information in the instance variables to correctly draw the picture. Obviously, that includes the coordinate arrays and the number of points that have been stored in the arrays. But the picture is different depending on whether or not the user has completed the polygon. If the polygon is complete, the picture shows a polygon; if not, the picture shows line segments connecting each point to the next. We need to record that basic distinction as part of the state. We can do that using a boolean instance variable, `complete`, which is true when a complete polygon should be drawn and `false` while the polygon is under construction. So, here are the necessary instance variables:

```
double[] xCoord, yCoord;  // Arrays to hold the
coordinates for up to 500 points.
int pointCt;              // Number of points in the
arrays.
```

```
boolean complete;          // Set to true when a polygon is
complete.
```

The arrays `xCoord` and `yCoord` are examples of partially full arrays (Subsection 3.8.4), although there is only one counter variable that applies to both arrays.

Given these instance variables, the `draw()` method can be written. To make the picture look nicer, I decided to use wide lines. There is also a question about what to draw when there is only one point. It's nice if the user can see that the point has been added to the data, but one point is not enough to draw a line or polygon. I decided to draw a small square at the first point, just to make it visible and noticeable. This also helps the user to find the first point when she is trying to click on it to complete the polygon. The fill color of the polygon is given by a global constant, `POLYGON_COLOR`.

```
private void draw() {
    GraphicsContext g = canvas.getGraphicsContext2D();
    g.setFill(Color.WHITE);
    g.fillRect(0,0,canvas.getWidth(),canvas.getHeight());
    if (pointCt == 0)
        return;
    g.setLineWidth(2);
    g.setStroke(Color.BLACK);
    if (complete) { // draw a polygon
        g.setFill(POLYGON_COLOR);
        g.fillPolygon(xCoord, yCoord, pointCt);
        g.strokePolygon(xCoord, yCoord, pointCt);
    }
    else { // show the lines the user has drawn so far
        g.setFill(Color.BLACK);
        g.fillRect(xCoord[0]-2, yCoord[0]-2, 4, 4);  //
small square marks first point
        for (int i = 0; i < pointCt - 1; i++) {
            g.strokeLine( xCoord[i], yCoord[i],
xCoord[i+1], yCoord[i+1]);
        }
    }
}
```

The rest of the program logic is in a `mousePressed()` routine that is called when the user presses the mouse on the canvas. In that method, state variables have to change in response to the user's actions. It requires some

care to do things in the right order. The three things that can happen are that the user starts a new polygon, the user completes the current polygon, or the user just adds a point to the current polygon. The conditions should be tested in that order:

```
if the current polygon is complete
    start a new polygon with the point where the user
clicked
else if the user clicked near the starting point
    complete the current polygon
else
    add the point that the user clicked to the data
call draw() to make the change visible
```

Actually, in my solution, I decided to add another case: The polygon can also be completed by right-clicking, or—to be very safe—if the number of points has reached 500. Also, there is a **bug** in the algorithm as stated, where it tests "if the user clicked near the first point". This test doesn't make sense unless there actually is a first point, that is unless `pointCt` is greater than zero. The test should really read "if `pointCt` > 0 and if the user clicked near `(xCoord[0],yCoord[0])`."

To complete a polygon just means setting the value of the variable `complete` to true. When `draw()` is called, the data will be displayed as a polygon. When a new polygon is started, the value of the variable `complete` has to be reset to false. Also, the first point on the polygon has to be put into the coordinate arrays, and `pointCt` must be set to 1 to indicate that there is only one point in the data. This much is pretty straightforward to implement.

The only thing in the algorithm that still needs implementing is to test whether the user clicks "near the starting point". The starting point has coordinates `(xCoord[0],yCoord[0])` and the point where the user clicked has coordinates `(evt.getX(),evt.getY())`. In my program, I check whether the `x`-coordinates of these points are three pixels or less apart and the `y`-coordinates are also three pixels or less apart. This is done by checking whether "`Math.abs(xCoord[0] - evt.getX()) <= 3 && Math.abs(yCoord[0] - evt.getY()) <= 3`".

The Solution

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.StackPane;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.input.MouseEvent;
import javafx.scene.input.MouseButton;
import javafx.scene.paint.Color;

/**
 * This program lets the user draw filled polygons.
 * The user inputs a polygon by clicking a series of
points.
 * The points are connected with lines from each point to
the
 * next Clicking near the starting point (within 3 pixels)
or
 * right-clicking will complete the polygon, so the user
can
 * begin a new one.  As soon as the user begins drawing a
new
 * polygon, the old one is discarded.
 */
public class SimplePolygons extends Application {

    public static void main(String[] args) {
        launch();
    }

    //-------------------------------------------------------
----------------

    private Canvas canvas;

    /* Variables for implementing polygon input. */

    private double[] xCoord, yCoord;    // Arrays
containing the points of
                                        //   the polygon.
Up to 500 points
                                        //   are allowed.

    private int pointCt;  // The number of points that have
been input.
```

```
    private boolean complete;   // Set to true when the
polygon is complete.
                                // When this is false, only
a series of lines are drawn.
                                // When it is true, a
filled polygon is drawn.

    private final static Color POLYGON_COLOR = Color.RED;
                                // Color that is used to
draw the polygons.


    /**
     * Set up the GUI, and install a mouse hander its data.
     */
    public void start (Stage stage) {

        xCoord = new double[500];  // create arrays to hold
the polygon's points
        yCoord = new double[500];
        pointCt = 0;

        canvas = new Canvas(400,400);
        draw();
        canvas.setOnMousePressed( e -> mousePressed(e) );

        StackPane root = new StackPane(canvas);
        root.setStyle("-fx-border-color: black");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Polygons");
        stage.setResizable(false);
        stage.show();

    }


    /**
     * Fill the canvas with white.  If the polygon is
complete, draw it.
     * If not, draw the lines that the user has input so
far.  (If only
     * one point has been input, it will still be visible
as a small dot.)
     */
    private void draw() {
        GraphicsContext g = canvas.getGraphicsContext2D();
```

```
        g.setFill(Color.WHITE);

g.fillRect(0,0,canvas.getWidth(),canvas.getHeight());
        if (pointCt == 0)
            return;
        g.setLineWidth(2);
        g.setStroke(Color.BLACK);
        if (complete) { // draw a polygon
            g.setFill(POLYGON_COLOR);
            g.fillPolygon(xCoord, yCoord, pointCt);
            g.strokePolygon(xCoord, yCoord, pointCt);
        }
        else { // show the lines the user has drawn so far
            g.setFill(Color.BLACK);
            g.fillRect(xCoord[0]-2, yCoord[0]-2, 4, 4);  //
small square marks first point
            for (int i = 0; i < pointCt - 1; i++) {
                g.strokeLine( xCoord[i], yCoord[i],
xCoord[i+1], yCoord[i+1]);
            }
        }
    }


    /**
     * Processes a mouse click.
     */
    private void mousePressed(MouseEvent evt) {

        if (complete) {
                // Start a new polygon at the point that
was clicked.
            complete = false;
            xCoord[0] = evt.getX();
            yCoord[0] = evt.getY();
            pointCt = 1;
        }
        else if ( pointCt > 0 && pointCt > 0 &&
(Math.abs(xCoord[0] - evt.getX()) <= 3)
                   && (Math.abs(yCoord[0] - evt.getY()) <=
3) ) {
                // User has clicked near the starting
point.
                // The polygon is complete.
            complete = true;
        }
        else if (evt.getButton() == MouseButton.SECONDARY
|| pointCt == 500) {
```

```
                // The polygon is complete.
            complete = true;
        }
        else {
                // Add the point where the user clicked to
the list of
                // points in the polygon, and draw a line
between the
                // previous point and the current point.  A
line can
                // only be drawn if there are at least two
points.
            xCoord[pointCt] = evt.getX();
            yCoord[pointCt] = evt.getY();
            pointCt++;
        }
        draw();  // in all cases, redraw the picture.
    } // end mousePressed()


}  // end class SimplePolygons
```

Solution for Programming Exercise 6.10

Exercise 6.10:

Write a GUI Blackjack program that lets the user play a game of Blackjack, with the computer as the dealer. The program should draw the user's cards and the dealer's cards, just as was done for the graphical HighLow card game in Subsection 6.6.1. 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 program and the Blackjack program 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 make the canvas just wide enough to show five cards, and tall enough to show both 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. It will also need the images file cards.png, which contains pictures of the cards.

The next exercise has a picture of a Blackjack game that you can use a guide, except that the version for this exercise does not allow betting. (Some

aesthetic changes to the GUI were made in that Blackjack program, compared to the HighLow program.)

Discussion

The `start()` method for this exercise can be very similar to that in the HighLow game. Aside from some tweaks to appearance, the canvas has to be bigger and the text of the buttons just has to be changed from "Higher" and "Lower" to "Hit" and "Stand". (The tweaks are discussed in the solution to the next exercise; they are not really needed in this version of Blackjack.)

In the HighLow game, there is one "hand," which holds all the cards that have been dealt. Blackjack is a two-player game, so there are two hands, one for the player and one for the dealer. These hands are of type `BlackjackHand`. So, we need instance variables

```
BlackjackHand dealerHand;   // The dealer's cards.
BlackjackHand playerHand;   // The user's cards.
```

We also need a `deck` of cards and a boolean-valued instance variable, `gameInProgress`, to keep track of the two basic states of the game: Is a game in progress, or are we between games? Finally, there is a `message` variable, which holds the string that is shown at the bottom of the game board.

There is a `drawBoard()` method that completely redraws the canvas, based on the current state of the game. It uses the information in the `dealerHand`, `playerHand`, `message`, and `gameInProgress` variables. The reason it needs to look at the `gameInProgress` variable is that when a game is in progress, one of the dealer's cards is drawn face down, so the user can't see it. Once the game is over, the card is drawn face up so the user can see what the dealer was holding. Note that there is no point in the program where I say, "turn the dealer's first card face up"! It happens automatically because the state of the game changes, and the `drawBoard()` method checks the state when it draws the canvas. If the game is over, the card is face up. If the game is in progress, the card is face down. This is nice example of state-machine thinking.

Note that writing the `drawBoard()` method required some calculation. The cards are 79 pixels wide and 123 pixels tall. Horizontally, there is a gap of 20 pixels between cards, and there are gaps of 20 pixels between the cards and the left and right edges. The total width needed for the canvas, 515, allows for five 79-pixel cards and six 20-pixel gaps: 5*79 + 6*20 = 515. The `N`-th card, counting from 0, has its left edge at `20+99*N`. It might be easier to see this as `20+79*N+20*N`, 20 pixels on the left plus `N` 79-pixel cards, plus a 20-pixel gap after each of the `N` cards. The vertical placement of the cards and strings was more experimental. I placed the dealer's hand at y-coordinate 40 and the player's hand at y-coordinate 206, leaving space to draw the strings "Dealer's Cards" and "Your Cards" above the hands. The message to the user is placed with its baseline 20 pixels above the bottom of the canvas, and I adjusted the height of the canvas to get the spacing right.

In this GUI version of Blackjack, things happen when the user clicks the "Hit", "Stand", and "New Game" buttons. The program handles these events by calling the methods `doHit()`, `doStand()`, and `doNewGame()`. Each of these methods has responsibility for one part of the game of Blackjack. Note that each method starts by checking the state of the game to make sure that it is legal to call the routine at this time. If `gameInProgress` is true, the user can legally click "Hit" or "Stand". If `gameInProgress` is false, the user can legally click "New Game". If the user made an illegal move, an error message is stored in the `message` variable, and `drawBoard()` is called so the user will see the new message. This is similar to the way the three buttons in HighLowGUI are handled.

The `doNewGame()` routine has to set up a new game. This means creating the deck and hands, shuffling the deck and dealing two cards into each hand. At this point, the first time I wrote the game, I just set `gameInProgress` to true, to record the fact that the state of the game has changed. Later, I realized that the `doNewGame()` routine also has to check whether one of the players has Blackjack, since there is really no other place where this can be done. It has to happen immediately after the first two cards are dealt. If one of the players has Blackjack, the game is over as soon as it starts, so `gameInProgress` has to be false, and the only action that the user can take at that point is to click the "New Game" button again. (Note that the `doNewGame()` routine is also

called by the `start()` method. This sets up the first game, so the user doesn't have to click on the "New Game" button to start the first game.)

When the user clicks "Hit", if the game is in progress, we deal a card into the user's hand. At this point, the state of the game might have changed. If the user has over 21, the user loses and the game is over. If the user has taken 5 cards without going over 21, the user wins and the game is over. In either of these cases, the value of the state variable `gameInProgress` becomes `false`. Otherwise, `gameInProgress` retains the value `true`, and the game will continue. Since `gameInProgress` is `true`, the user still has the choice of clicking "Hit" or "Stand". (Note that there is no loop in the program that says "while the user continues to hit." The progress of the game is driven by events.)

Finally, when the user clicks "Stand", the game is definitely over, so `gameInProgress` is set to false. However, before the game can end, the dealer gets to draw cards and a winner is determined. This all has to be done in the `doStand()` routine. Then, the canvas is redrawn to show the final state of the game.

The Solution

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;
import javafx.geometry.Pos;
import javafx.scene.control.Button;
import javafx.scene.image.Image;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;

/**
 * In this program, the user plays a game of Blackjack.
The
 * computer acts as the dealer.  The user plays by clicking
 * "Hit!" and "Stand!" buttons.
 *
```

```
 * This program depends on the following classes:  Card,
Hand,
 * BlackjackHand, Deck.  It also requires the image
resource
 * file cards.png.
 */
public class BlackjackGUI extends Application {

    public static void main(String[] args) {
        launch(args);
    }

    //-------------------------------------------------------
----------------

    private Deck deck;         // A deck of cards to be
used in the game.

    private BlackjackHand dealerHand;   // Hand containing
the dealer's cards.
    private BlackjackHand playerHand;   // Hand containing
the user's cards.

    private String message; // A message drawn on the
canvas, which changes
                            //    to reflect the state of
the game.

    private boolean gameInProgress; // Set to true when a
game begins and to false
                                    //   when the game
ends.

    private Canvas board;     // The canvas were cards and
messages are displayed.

    private Image cardImages;  // The image that contains
all the cards in a deck.


    /**
     * The start() method() sets up the GUI and event
handling.
     */
    public void start(Stage stage) {

        cardImages = new Image("cards.png");
```

```
        board = new Canvas(515, 390);
                            // space for 5 cards across and 2
cards down,
                            // with 20-pixel spaces between
cards,
                            // plus space for messages

        Button hitButton = new Button( "Hit!" );
        hitButton.setOnAction( e -> doHit() );
        Button standButton = new Button( "Stand!" );
        standButton.setOnAction( e -> doStand() );
        Button newGameButton = new Button( "New Game" );
        newGameButton.setOnAction( e -> doNewGame() );

        HBox buttonBar = new
HBox(6,hitButton,standButton,newGameButton);
        buttonBar.setStyle("-fx-border-color: darkred; -fx-
border-width: 3px 0 0 0;"
                + "-fx-padding: 8px; -fx-background-
color:beige");
        buttonBar.setAlignment(Pos.CENTER);

        BorderPane root = new BorderPane();
        root.setStyle("-fx-border-color: darkred; -fx-
border-width: 3px");
        root.setCenter(board);
        root.setBottom(buttonBar);

        doNewGame(); // Start the first game.

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Blackjack");
        stage.setResizable(false);
        stage.show();

    }  // end start()


    /**
     * This method is called when the user clicks the
"Hit!" button.  First
     * check that a game is actually in progress.  If not,
give  an error
     * message and exit.  Otherwise, give the user a card.
The game can end
     * at this point if the user goes over 21 or if the
user has taken 5 cards
```

```
     * without going over 21.
     */
    void doHit() {
        if (gameInProgress == false) {
            message = "Click \"New Game\" to start a new
game.";
            drawBoard();
            return;
        }
        playerHand.addCard( deck.dealCard() );
        if ( playerHand.getBlackjackValue() > 21 ) {
            message = "You've busted!  Sorry, you lose.";
            gameInProgress = false;
        }
        else if (playerHand.getCardCount() == 5) {
            message = "You win by taking 5 cards without
going over 21.";
            gameInProgress = false;
        }
        else {
            message = "You have " +
playerHand.getBlackjackValue() + ".  Hit or Stand?";
        }
        drawBoard();
    }


    /**
     * This method is called when the user clicks the
"Stand!" button.
     * Check whether a game is actually in progress.  If it
is, the game
     * ends.  The dealer takes cards until either the
dealer has 5 cards
     * or more than 16 points.  Then the  winner of the
game is determined.
     */
    void doStand() {
        if (gameInProgress == false) {
            message = "Click \"New Game\" to start a new
game.";
            drawBoard();
            return;
        }
        gameInProgress = false;
        while (dealerHand.getBlackjackValue() <= 16 &&
dealerHand.getCardCount() < 5)
            dealerHand.addCard( deck.dealCard() );
```

```
        if (dealerHand.getBlackjackValue() > 21)
            message = "You win!  Dealer has busted with " +
dealerHand.getBlackjackValue() + ".";
        else if (dealerHand.getCardCount() == 5)
            message = "Sorry, you lose.  Dealer took 5
cards without going over 21.";
        else if (dealerHand.getBlackjackValue() >
playerHand.getBlackjackValue())
            message = "Sorry, you lose, " +
dealerHand.getBlackjackValue()
                                + " to " +
playerHand.getBlackjackValue() + ".";
        else if (dealerHand.getBlackjackValue() ==
playerHand.getBlackjackValue())
            message = "Sorry, you lose.  Dealer wins on a
tie.";
        else
            message = "You win, " +
playerHand.getBlackjackValue()
                                + " to " +
dealerHand.getBlackjackValue() + "!";
        drawBoard();
    }


    /**
     * Called by the constructor, and called by
doNewGame().  Start a new game.
     * Deal two cards to each player.  The game might end
right then  if one
     * of the players had blackjack.  Otherwise,
gameInProgress is set to true
     * and the game begins.
     */
    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!";
            drawBoard();
            return;
        }
        deck = new Deck();   // Create the deck and hands
to use for this game.
        dealerHand = new BlackjackHand();
        playerHand = new BlackjackHand();
```

```
        deck.shuffle();
        dealerHand.addCard( deck.dealCard() );  // Deal two
cards to each player.
        dealerHand.addCard( deck.dealCard() );
        playerHand.addCard( deck.dealCard() );
        playerHand.addCard( deck.dealCard() );
        if (dealerHand.getBlackjackValue() == 21) {
            message = "Sorry, you lose.  Dealer has
Blackjack.";
            gameInProgress = false;
        }
        else if (playerHand.getBlackjackValue() == 21) {
            message = "You win!  You have Blackjack.";
            gameInProgress = false;
        }
        else {
            message = "You have " +
playerHand.getBlackjackValue() + ".  Hit or stand?";
            gameInProgress = true;
        }
        drawBoard();
    }  // end newGame();


    /**
     * The drawBoard() method shows the message at the
bottom of the
     * canvas, and it draws all of the dealt cards spread
out
     * across the canvas.
     */
    public void drawBoard() {

        GraphicsContext g = board.getGraphicsContext2D();
        g.setFill( Color.DARKGREEN);
        g.fillRect(0,0,board.getWidth(),board.getHeight());

        g.setFont( Font.font(16) );
        g.setFill( Color.rgb(220,255,220) );

        // Draw the message at the bottom of the canvas.

        g.fillText(message, 20, board.getHeight() - 20);

        // Draw labels for the two sets of cards.

        g.fillText("Dealer's Cards:", 20, 27);
        g.fillText("Your Cards:", 20, 190);
```

```
        // Draw dealer's cards.  Draw first card face down
if
        // the game is still in progress,  It will be
revealed
        // when the game ends.

        if (gameInProgress)
            drawCard(g, null, 20, 40);
        else
            drawCard(g, dealerHand.getCard(0), 20, 40);
        for (int i = 1; i < dealerHand.getCardCount(); i++)
            drawCard(g, dealerHand.getCard(i), 20 + i * 99,
40);

        // Draw the user's cards.

        for (int i = 0; i < playerHand.getCardCount(); i++)
            drawCard(g, playerHand.getCard(i), 20 + i * 99,
206);

    }  // end drawBoard();


    /**
     * Draws a card with top-left corner at (x,y).  If card
is null,
     * then a face-down card is drawn.  The cards images
are from
     * the file cards.png; this program will fail without
it.
     */
    private void drawCard(GraphicsContext g, Card card, int
x, int y) {
        int cardRow, cardCol;
        if (card == null) {
            cardRow = 4;   // row and column of a face down
card
            cardCol = 2;
        }
        else {
            cardRow = 3 - card.getSuit();
            cardCol = card.getValue() - 1;
        }
        double sx,sy;  // top left corner of source rect
for card in cardImages
        sx = 79 * cardCol;
        sy = 123 * cardRow;
```

```
        g.drawImage( cardImages, sx,sy,79,123, x,y,79,123
);
    } // end drawCard()

} // end class BlackjackGUI
```

Solution for Programming Exercise 6.11

Exercise 6.11:

In the Blackjack game from Exercise 6.10, 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 method that can be called every time it is necessary to set the value of the `gameInProgress` variable. That method can take full responsibility for enabling and disabling the buttons (as long as it is used consistently). Recall that if `bttn` is a variable of type `Button`, then `bttn.setDisable(true)` disables the button and `bttn.setDisable(false)` 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 program starts, give the user $100. Add a *TextField* to the strip of controls along the bottom of the panel. The user enters the bet in this *TextField*. 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 *TextField* might not be a legal number, the bet that the user places might be more money than the user has, or the bet 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 *TextField* uneditable while the game is in progress. If `betInput` is the *TextField*, you can make it editable and uneditable by the user with the commands `betInput.setEditable(true)` and `betInput.setEditable(false)`.

In the `drawBoard()` 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 program 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 `start()` method, 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:

Discussion

First, I want to briefly discuss the changes that I made to the appearance of the user interface, compared to the program, HighLowGUI.java, on which the Blackjack programs are based.

For Blackjack, I added a dark red border around the *BorderPane* that serves as the `root` node of the scene graph. I also wanted a red line between the canvas and the `buttonBar` that is in the bottom position. For that, I added a

border to `buttonBar`, but only on its top edge; the border width on the other three sides of the `buttonBar` is set to zero.

In *HighLowGUI*, the three buttons are all the same size, and they fill the `buttonBar`. In the final version of Blackjack, the button bar contains five components, and it didn't look good to make them all the same width. Furthermore, I wanted to give a nicer background color (beige) to the button bar, and it needed some padding to add some space between the components and the edges of the button bar. Finally, to put the components that it contains in the center of the button bar rather than at the left end, the alignment property of the button bar is set to `Pos.CENTER`. So, in my Blackjack program, the root node and button bar are created and configured in the `start()` method as follows:

```
HBox buttonBar = new HBox(6, hitButton, standButton,
newGameButton,
        new Label(" Your bet:"), betInput);
buttonBar.setStyle("-fx-border-color: darkred; -fx-border-
width: 3px 0 0 0;"
        + "-fx-padding: 8px; -fx-background-color:beige");
buttonBar.setAlignment(Pos.CENTER);

BorderPane root = new BorderPane();
root.setStyle("-fx-border-color: darkred; -fx-border-
width: 3px");
root.setCenter(board);
root.setBottom(buttonBar);
```

All of these details might seem very complicated, but it does get easier with practice!

The buttons in the program must be enabled and disabled whenever the value of the variable `gameInProgress` changes. At the same time, the text field should be made editable or non-editable. As recommended in the exercise, I wrote a method for changing the value of `gameInProgress`. This method also sets the buttons and text field to reflect the state of the program, which allows the state of the buttons and text field to be controlled in one location:

```
/**
 * This method is called whenever the value of the
```

```
gameInProgress
 * property has to be changed.  In addition to setting the
value
 * of the gameInProgress variable, it also enables and
disables
 * the buttons and text input box to reflect the state of
the game.
 * @param inProgress The new value of gameInProgress.
 */
private void setGameInProgress( boolean inProgress ) {
    gameInProgress = inProgress;
    if (gameInProgress) {
        hitButton.setDisable(false);
        standButton.setDisable(false);
        newGameButton.setDisable(true);
        betInput.setEditable(false);
        hitButton.requestFocus();
    }
    else {
        hitButton.setDisable(true);
        standButton.setDisable(true);
        newGameButton.setDisable(false);
        betInput.setEditable(true);
        newGameButton.requestFocus();
    }
}
```

Note that this method requests that the "Hit" button get focus of there is a game in progress, and that the "New Game" button get focus if not. When a button has the input focus, the user can trigger the button by pressing the space bar (or the Enter key). The idea is to give the input focus to the button that represents the user's most likely next action. This makes it possible for the user to play the game largely by hitting the space bar, instead of using the mouse. (In fact, since the TAB key can be used to move the input focus, the user can play the game without ever touching the mouse.)

Once `setGameInProgress()` is available, then any line in the old program that said "`gameInProgress = false;`" should be changed to "`setGameInProgress(false);`". And any line that said "`gameInProgress = true;`" should be changed to "`setGameInProgress(true);`". In this way, we can be sure that the buttons are always properly enabled and disabled. Note that I added a call to `setGameInProgress(false)` to the `start()` method as well, to make sure

the buttons and textfield are properly configured at the beginning of the program. Since the new program does not immediately start the first game, `gameInProgress` should be set to `false` in the `start()` method. (Also, the call to `doNewGame()` in the `start()` method was deleted).

You should understand why I use a subroutine to set the value of `gameInProgress`. Every time `gameInProgress` changes, each of the buttons has to be enabled or disabled and the text field has to be made editable or uneditable. That's four extra lines of code each time the program says `gameInProgress = true` or `gameInProgress = false`. We can avoid some extra typing by calling the subroutine. Furthermore, if we always call the subroutine to set the value of `gameInProgress`, we can be sure that the states of the buttons and text field will always be set correctly to match the value of `gameInProgress`.

The changes that I've discussed so far are enough to complete the first part of the exercise, enabling and disabling the buttons. We still have to implement betting.

I added several new instance variables to the program to implement betting: a *TextField*, `betInput`, where the user inputs the bet amount; an int variable, `usersMoney`, to hold the number of dollars that the user currently has; and another int variable, `betAmount`, to store the amount of the user's bet while a game is in progress.

The *TextField* for the user's input is created with the command "`betInput = new TextField("10");`". The parameter in the constructor specifies the initial content of the text input box. This is meant as a reasonable value for the bet, but the user can change it if she wants to. The number of preferred columns for the textfield is reduced by calling `betInput.setPrefColumnCount(5)`. The preferred size of the *TextField* is computed based on the preferred column count. Without this change, the textfield would be too big for this program.

The value of `usersMoney` is initialized to 100 when it is declared. Any time a game ends, if the user wins, then `betAmount` is added to the user's money, and if the user loses, then `betAmount` is subtracted from the user's money. We

have to decide what happens if the user runs out of money. One possibility would be to shut the game down, but that seems drastic since it's only play money anyway. So, if the value of `usersMoney` drops to zero, I give the user another $100 by setting the value of `usersMoney` back to 100; this is done in `doNewGame()` at the start of the next game.

At the beginning of a game in `doNewGame()`, the program has to look at the number in the text field to determine how much money the user wants to bet on the game. Several things can go wrong at this time: The text in the box might not be a legal integer, the amount might be more money than the user has, and the value might be less than or equal to zero. Each of these possibilities has to be tested, and, if an error is detected, an error message has to be shown to the user and no new game is started. It's a little complicated:

```
try {  // get the amount of the user's bet and check for
errors
    betAmount = Integer.parseInt(betInput.getText());
}
catch (NumberFormatException e) {
    message = "Bet amount must be an integer!";
    betInput.requestFocus();
    betInput.selectAll();
    drawBoard();
    return;
}
if (betAmount > usersMoney) {
    message = "The bet amount can't be more than you
have!";
    betInput.requestFocus();
    betInput.selectAll();
    drawBoard();
    return;
}
if (betAmount <= 0) {
    message = "The bet has to be a positive number";
    betInput.requestFocus();
    betInput.selectAll();
    drawBoard();
    return;
}
```

If the program gets past all of these tests, it starts the game as it did in the old program.

When the game ends for any reason, the user's money has to be adjusted. There are many points in the source code where the game ends. In each of those places, I inserted a line "`usersMoney = usersMoney + betAmount`" if the user won or "`usersMoney = usersMoney - betAmount`" if the user lost. It might have been a good idea to write another subroutine to handle this task.

I also added some code to the `drawBoard()` method to display the user's current amount of money. To accommodate this, this version of the panel is 25 pixels taller than the previous version.

If the first game has not started, `drawBoard()` draws a welcome message instead of showing the user's and dealer's hands. There is the question of how to test whether the first game has started. The obvious (and perhaps best) way to handle that would have been to add another boolean state variable such as `firstGameHasStarted`. However, I used the fact that `dealerHand` is `null` before the first game starts (and only then). So `drawBoard()` can test whether `dealerHand` is `null` to decide whether to draw the welcome message.

The Solution

Here is the new version of the Blackjack source code, with changes from Exercise 6.10 shown in red italic:

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;
import javafx.geometry.Pos;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.image.Image;
import javafx.scene.canvas.Canvas;
```

```
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;

/**
 * In this program, the user plays a game of Blackjack.
The
 * computer acts as the dealer.  The user plays by clicking
 * "Hit!" and "Stand!" buttons.  The user can place bets.
 * At the beginning of the game, the user is given $100.
 *
 * This program depends on the following classes:  Card,
Hand,
 * BlackjackHand, Deck.  It also requires the image
resource
 * file cards.png.
 */
public class BlackjackGUI2 extends Application {

    public static void main(String[] args) {
        launch();
    }

    //-------------------------------------------------------
----------------

    private Deck deck;         // A deck of cards to be
used in the game.

    private BlackjackHand dealerHand;   // Hand containing
the dealer's cards.
    private BlackjackHand playerHand;   // Hand containing
the user's cards.

    private Button hitButton, standButton, newGameButton;

    private TextField betInput;  // An input box for the
user's bet amount.

    private String message; // A message drawn on the
canvas, which changes
                            //    to reflect the state of
the game.

    private boolean gameInProgress; // Set to true when a
game begins and to false
                                    //   when the game
```

```
ends.

    private Canvas board;     // The canvas were cards and
messages are displayed.

    private Image cardImages;  // The image that contains
all the cards in a deck.

    private int usersMoney = 100;  // How much money the
user currently has.

    private int betAmount;  // The amount the use bet on
the current game,
                            //      when a game is in
progress.

    /**
     * The start() method() sets up the GUI and event
handling.
     */
    public void start(Stage stage) {

        cardImages = new Image("cards.png");

        board = new Canvas(515, 415);
            // space for 5 cards across and 2 cards down,
            // with 20-pixel spaces between cards,
            // plus space for messages

        hitButton = new Button( "Hit!" );
        hitButton.setOnAction( e -> doHit() );
        standButton = new Button( "Stand!" );
        standButton.setOnAction( e -> doStand() );
        newGameButton = new Button( "New Game" );
        newGameButton.setOnAction( e -> doNewGame() );

        betInput = new TextField("10");
        betInput.setPrefColumnCount(5);

        HBox buttonBar = new HBox(6, hitButton,
standButton, newGameButton,
                new Label(" Your bet:"), betInput);
        buttonBar.setStyle("-fx-border-color: darkred; -fx-
border-width: 3px 0 0 0;"
                + "-fx-padding: 8px; -fx-background-
color:beige");
        buttonBar.setAlignment(Pos.CENTER);
```

```
        BorderPane root = new BorderPane();
        root.setStyle("-fx-border-color: darkred; -fx-
border-width: 3px");
        root.setCenter(board);
        root.setBottom(buttonBar);

        setGameInProgress(false);
        drawBoard();

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Blackjack");
        stage.setResizable(false);
        stage.show();

    }  // end start()


    /**
     * This method is called whenever the value of the
gameInProgress
     * property has to be changed.  In addition to setting
the value
     * of the gameInProgress variable, it also enables and
disables
     * the buttons and text input box to reflect the state
of the game.
     * @param inProgress The new value of gameInProgress.
     */
    private void setGameInProgress( boolean inProgress ) {
        gameInProgress = inProgress;
        if (gameInProgress) {
            hitButton.setDisable(false);
            standButton.setDisable(false);
            newGameButton.setDisable(true);
            betInput.setEditable(false);
            hitButton.requestFocus();
        }
        else {
            hitButton.setDisable(true);
            standButton.setDisable(true);
            newGameButton.setDisable(false);
            betInput.setEditable(true);
            newGameButton.requestFocus();
        }
```

```
    }


    /**
     * This method is called when the user clicks the
"Hit!" button.  First
     * check that a game is actually in progress.  If not,
give  an error
     * message and exit.  Otherwise, give the user a card.
The game can end
     * at this point if the user goes over 21 or if the
user has taken 5 cards
     * without going over 21.
     */
    void doHit() {
        if (gameInProgress == false) {
                // Should be impossible, since the Hit
button
                // is disabled when it is not legal to use
it.
            message = "Click \"New Game\" to start a new
game.";
            drawBoard();
            return;
        }
        playerHand.addCard( deck.dealCard() );
        if ( playerHand.getBlackjackValue() > 21 ) {
            usersMoney = usersMoney - betAmount;
            message = "You've busted!  Sorry, you lose.";
            setGameInProgress(false);
        }
        else if (playerHand.getCardCount() == 5) {
            usersMoney = usersMoney + betAmount;
            message = "You win by taking 5 cards without
going over 21.";
            setGameInProgress(false);
        }
        else {
            message = "You have " +
playerHand.getBlackjackValue() + ".  Hit or Stand?";
        }
        drawBoard();
    }


    /**
     * This method is called when the user clicks the
```

```
"Stand!" button.
     * Check whether a game is actually in progress.  If it
is, the game
     * ends.  The dealer takes cards until either the
dealer has 5 cards
     * or more than 16 points.  Then the  winner of the
game is determined.
     */
    void doStand() {
        if (gameInProgress == false) {
                // Should be impossible, since the Stand
button
                // is disabled when it is not legal to use
it.
            message = "Click \"New Game\" to start a new
game.";
            drawBoard();
            return;
        }
        setGameInProgress(false);
        while (dealerHand.getBlackjackValue() <= 16 &&
dealerHand.getCardCount() < 5)
            dealerHand.addCard( deck.dealCard() );
        if (dealerHand.getBlackjackValue() > 21) {
            usersMoney = usersMoney + betAmount;
            message = "You win!  Dealer has busted with " +
dealerHand.getBlackjackValue() + ".";
        }
        else if (dealerHand.getCardCount() == 5) {
            usersMoney = usersMoney - betAmount;
            message = "Sorry, you lose.  Dealer took 5
cards without going over 21.";
        }
        else if (dealerHand.getBlackjackValue() >
playerHand.getBlackjackValue()) {
            usersMoney = usersMoney - betAmount;
            message = "Sorry, you lose, " +
dealerHand.getBlackjackValue()
                                    +    " to " +
playerHand.getBlackjackValue() + ".";
        }
        else if (dealerHand.getBlackjackValue() ==
playerHand.getBlackjackValue()) {
            usersMoney = usersMoney - betAmount;
            message = "Sorry, you lose.  Dealer wins on a
tie.";
        }
```

```
        else {
            usersMoney = usersMoney + betAmount;
            message = "You win, " +
playerHand.getBlackjackValue()
                            + " to " +
dealerHand.getBlackjackValue() + "!";
        }
        drawBoard();
    }


    /**
     * Called by the constructor, and called by
doNewGame().  Start a new game.
     * Deal two cards to each player.  The game might end
right then  if one
     * of the players had blackjack.  Otherwise,
gameInProgress is set to true
     * and the game begins.
     */
    void doNewGame() {
        if (gameInProgress) {
                // If the current game is not over, it is
an error to try
                // to start a new game.  Should be
impossible, since
                // the New Game button is disabled when it
is not legal to use it.
            message = "You still have to finish this
game!";
            drawBoard();
            return;
        }
        if (usersMoney == 0) { // User is broke; give the
user another $100.
            usersMoney = 100;
        }
        try {  // get the amount of the user's bet and
check for errors
            betAmount =
Integer.parseInt(betInput.getText());
        }
        catch (NumberFormatException e) {
            message = "Bet amount must be an integer!";
            betInput.requestFocus();
            betInput.selectAll();
```

```
            drawBoard();
            return;
        }
        if (betAmount > usersMoney) {
            message = "The bet amount can't be more than
you have!";
            betInput.requestFocus();
            betInput.selectAll();
            drawBoard();
            return;
        }
        if (betAmount <= 0) {
            message = "The bet has to be a positive
number";
            betInput.requestFocus();
            betInput.selectAll();
            drawBoard();
            return;
        }
        deck = new Deck();   // Create the deck and hands
to use for this game.
        dealerHand = new BlackjackHand();
        playerHand = new BlackjackHand();
        deck.shuffle();
        dealerHand.addCard( deck.dealCard() );  // Deal two
cards to each player.
        dealerHand.addCard( deck.dealCard() );
        playerHand.addCard( deck.dealCard() );
        playerHand.addCard( deck.dealCard() );
        if (dealerHand.getBlackjackValue() == 21) {
            message = "Sorry, you lose.  Dealer has
Blackjack.";
            usersMoney = usersMoney - betAmount;
            setGameInProgress(false);
        }
        else if (playerHand.getBlackjackValue() == 21) {
            message = "You win!  You have Blackjack.";
            usersMoney = usersMoney + betAmount;
            setGameInProgress(false);
        }
        else {
            message = "You have " +
playerHand.getBlackjackValue() + ".  Hit or stand?";
            setGameInProgress(true);
        }
        drawBoard();
```

```
    }  // end newGame();


    /**
     * The drawBoard() method shows the messages at the
bottom of the
     * canvas, and it draws all of the dealt cards spread
out
     * across the canvas.  If the first game has not
started, it shows
     * a welcome message instead of the cards.
     */
    public void drawBoard() {

        GraphicsContext g = board.getGraphicsContext2D();
        g.setFill( Color.DARKGREEN);
        g.fillRect(0,0,board.getWidth(),board.getHeight());

        g.setFont( Font.font(16) );

        // Draw a message telling how much money the user
has.

        g.setFill(Color.YELLOW);
        if (usersMoney > 0) {
            g.fillText("You have $" + usersMoney, 20,
board.getHeight() - 45);
        }
        else {
            g.fillText("YOU ARE BROKE!  (I will give you
another $100.)",
                        20, board.getHeight() - 45 );
            usersMoney = 100;
        }

        g.setFill( Color.rgb(220,255,220) );

        if (dealerHand == null) {
                // The first game has not yet started.
                // Draw a welcome message and return.
            g.setFont( Font.font(30) );
            g.fillText("Welcome to Blackjack!\nPlace your
bet and\nclick \"New Game\".", 40,80);
            return;
        }
```

```
        // Draw the message at the bottom of the canvas.

        g.fillText(message, 20, board.getHeight() - 20);

        // Draw labels for the two sets of cards.

        g.fillText("Dealer's Cards:", 20, 27);
        g.fillText("Your Cards:", 20, 190);

        // Draw dealer's cards.  Draw first card face down
if
        // the game is still in progress,  It will be
revealed
        // when the game ends.

        if (gameInProgress)
            drawCard(g, null, 20, 40);
        else
            drawCard(g, dealerHand.getCard(0), 20, 40);
        for (int i = 1; i < dealerHand.getCardCount(); i++)
            drawCard(g, dealerHand.getCard(i), 20 + i * 99,
40);

        // Draw the user's cards.

        for (int i = 0; i < playerHand.getCardCount(); i++)
            drawCard(g, playerHand.getCard(i), 20 + i * 99,
206);

    }  // end drawBoard();


    /**
     * Draws a card with top-left corner at (x,y).  If card
is null,
     * then a face-down card is drawn.  The cards images
are from
     * the file cards.png; this program will fail without
it.
     */
    private void drawCard(GraphicsContext g, Card card, int
x, int y) {
        int cardRow, cardCol;
        if (card == null) {
            cardRow = 4;   // row and column of a face down
card
            cardCol = 2;
        }
```

```
        else {
            cardRow = 3 - card.getSuit();
            cardCol = card.getValue() - 1;
        }
        double sx,sy;  // top left corner of source rect
for card in cardImages
        sx = 79 * cardCol;
        sy = 123 * cardRow;
        g.drawImage( cardImages, sx,sy,79,123, x,y,79,123
);
    } // end drawCard()


} // end class BlackjackGUI2
```

Solution for Programming Exercise 7.1

Exercise 7.1:

Write a subroutine that creates an ArrayList containing several *different* random integers in the range from 1 up to some specified maximum. The number of integers and the maximum allowed value for the integers should be parameters to the subroutine. Write a `main()` routine to test your subroutine.

Discussion

The exercise asks for an ArrayList containing integers. That has to mean using `ArrayList<Integer>`, since an ArrayList can't contain values of type int. The problem is pretty easy: Start with an empty list, then generate random integers and add them to the list until you have as many integers as you want. Since all the integers in the list must be different, before adding an integer to the list, we need to test whether that integer is already in the list. One way to do that is with the `indexOf()` method from the ArrayList class. If `numbers` is an `ArrayList<Integer>` and `num` is an int, then `numbers.indexOf(num)` returns -1 if `num` is **not** in the list. (Note that this relies on autoboxing to convert `num` into an object of type *Integer*, since `indexOf()` requires an object as its parameter.) To create the list of numbers, we just add integers to the list until it reaches the desired size. Assuming that `count` is the desired number of integers and `max` is the maximum allowed value for the integers, this can be done with

```
ArrayList<Integer> numbers = new ArrayList<Integer>();
while (numbers.size() < count) {
    int num = (int)(Math.random()*max) + 1;
    if ( numbers.indexOf(num) == -1 )
        numbers.add( num );
}
```

This code is the heart of the solution. There is one problem, however. If `count > max`, then it's impossible to get `count` different integers in the range

from 1 to `max`, because there aren't that many different numbers in the range! In fact, the `while` loop will be an infinite loop in that case since the size of the ArrayList can never become larger than `max`. My solution avoids the infinite loop by throwing an *IllegalArgumentException* when `count > max`. You can see the complete subroutine, and a `main()` routine that tests it, below.

Note that an ArrayList in this program is simply printed using `System.out.println()`. The `toString()` method of the *ArrayList* class will be called to get the *String* representation of the list. That method outputs the items from the list, enclosed between `[` and `]` and separated by commas.

The Solution

```
import java.util.ArrayList;


public class SeveralRandomIntegers {

    /**
     * Creates an ArrayList that contains several integers
chosen at
     * random from a specified range of values.  All the
integers
     * in the list are different.
     * @param count  the number of random integers to be
created
     * @param max  the integers are chosen in the range 1
to max, inclusive
     * @return an ArrayList containing the integers.  The
integers are
     *    random and are in a random order.
     * @throws IllegalArgumentException if max is greater
than count.  In
     *    that case there are fewer than count different
integers in the
     *    range 1 to max.
     */
    public static ArrayList<Integer> makeRandomInts( int
count, int max ) {
        if (count > max) {
            throw new IllegalArgumentException("Can't have
" + count +
```

```
                    " different integers in the range  1 to
" + max);
        }
        ArrayList<Integer> numbers = new ArrayList<Integer>
();
        while (numbers.size() < count) {
            int num = (int)(Math.random()*max) + 1;
            if ( numbers.indexOf(num) == -1 )
                numbers.add( num );
        }
        return numbers;
    }

    public static void main(String[] args) {
        System.out.println("10 integers chosen from the
range 1 to 100:");
        for (int i = 0; i < 8; i++)
            System.out.println( makeRandomInts(10,100) );
        System.out.println();
        System.out.println("10 integers chosen from the
range 1 to 10:");
        for (int i = 0; i < 3; i++)
            System.out.println( makeRandomInts(10,10) );
        System.out.println();
        System.out.println("25 integers chosen from the
range 1 to 10000:");
        System.out.println( makeRandomInts(25,10000) );
        System.out.println();
        System.out.println("End with an exception!");
        makeRandomInts(10,9);
    }

}
```

Solution for Programming Exercise 7.2

Exercise 7.2:

Suppose that `M` is a two-dimensional array that has `R` rows and `C` columns. The **transpose** of `M` is defined to be an array `T` that has `C` rows and `R` columns such that `T[i][j] = M[j][i]` for each `i` and `j`. Write a function that takes an array of type `int[][]` as a parameter, and returns the transpose of that array. (Assume that the parameter is a typical 2D array in which all the rows have the same length.) Also write a subroutine to print a 2D array of integers in neat rows and columns, and include a `main()` routine to test your work.

Discussion

To create the transpose, we need to know how many rows and how many columns are in the original array. As noted in Subsection 7.6.1, the number of rows in a 2D array `A` is given by `A.length`, and the number of columns is `A[0].length`. (`A[0].length` is the length of the first row of `A`, which is the same as the number of columns in `A`. Remember that all rows have the same length.) Given that, we can create a transpose array of the correct size, and—with a little care to get the indices right—we can copy all the items from the original array into the transpose. If `matrix` is the original array, then the transpose can be created with

```
int[][] transpose;
int R = matrix.length;      // the number of rows in matrix
int C = matrix[0].length;   // the number of columns in
matrix
transpose = new int[C][R];
for ( int i = 0; i < C; i++) { // goes through ROWS of the
transpose
    for ( int j = 0; j < R; j++ ) { // goes through
COLUMNS of the transpose
        transpose[i][j] = matrix[j][i];
    }
}
```

One other note of interest is in the code for printing the array. See the comments on the source code below.

By the way, one purpose of this exercise was to remind you that arrays can be parameters and they can be the return value of a function. In this case, the parameter type and the return type for the function are both given by `int[][]`.

The Solution

```
public class TransposeMatrix {

    /**
     * Creates the transpose of a given 2D array of
integers.
     * @param matrix  the original array
     * @return the transpose of matrix
     */
    public static int[][] computeTranspose( int[][] matrix
) {
        int[][] transpose;
        int R = matrix.length;     // the number of rows in
matrix
        int C = matrix[0].length;  // the number of columns
in matrix
        transpose = new int[C][R];
        for ( int i = 0; i < C; i++) { // goes through ROWS
of the transpose
            for ( int j = 0; j < R; j++ ) { // goes through
COLUMNS of the transpose
                transpose[i][j] = matrix[j][i];
            }
        }
        return transpose;
    }

    /**
     * Prints out the items of a 2D array of ints in rows
and columns,
     * with 6 spaces in each column.
     */
    public static void print( int[][] array ) {
            // Note that this uses a for-each loop where
the loop
```

```
            // control variable is of type int[].  This
works because
            // a 2D array is actually a 1D array of 1D
arrays, where
            // each 1D array is one of the rows of the 2D
array.
        for ( int[] row : array ) {
                // print out one row from the array
            System.out.print("   ");
            for ( int item : row ) {
                   // print with 1 blank space and 5 spaces
for the integer;
                   // if an integer needs more than 5
spaces, the columns will
                   // be messed up, but all the integers
will still be readable.
                System.out.printf(" %5d", item);
            }
            System.out.println();
        }
    }

    /**
     * Test the subroutines by creating two arrays and
printing them and
     * their transposes.  The arrays are constructed so
that it is easy
     * to see that the transposes are correct.
     */
    public static void main(String[] args) {
        int[][] orig = {
                { 1, 2, 3, 4, 5, 6 },
                { 10, 20, 30, 40, 50, 60 },
                { 100, 200, 300, 400, 500, 600 }
        };
        System.out.println("Original matrix:");
        System.out.println();
        print(orig);
        System.out.println();
        System.out.println("The transpose:");
        System.out.println();
        print( computeTranspose(orig) );
        System.out.println();
        System.out.println();

        orig = new int[][] {
                {1, 1, 1, 1, 1, 1, 1},
                {2, 2, 2, 2, 2, 2, 2},
```

```
                {3, 3, 3, 3, 3, 3, 3},
                {4, 4, 4, 4, 4, 4, 4},
                {5, 5, 5, 5, 5, 5, 5},
                {6, 6, 6, 6, 6, 6, 6},
                {7, 7, 7, 7, 7, 7, 7},
        };
        System.out.println("Original matrix:");
        System.out.println();
        print(orig);
        System.out.println();
        System.out.println("The transpose:");
        System.out.println();
        print( computeTranspose(orig) );
        System.out.println();

    }

}
```

Solution for Programming Exercise 7.3

Exercise 7.3:

In Subsection 7.5.4, it is mentioned that the standard sorting method `Arrays.sort()` is much faster and efficient than selection sort. Write a program to test this claim. To be specific, your program should create a large array filled with random real numbers. It should use both `Arrays.sort()` and `selectionSort()` to sort the array, and it should time how long it takes to perform each sort. Furthermore, it should do the same thing for a large array of random *Strings*. To find the times, you can use `System.nanoTime()` (see Subsection 2.3.1 and the example TimedComputation.java).

Discussion

This exercise is most interesting for the results of the timing experiments, but one point of interest is how to make a "random string." In my program, I make a string containing uppercase letters, with a random length:

```
private static String randomString() {
    int length = 5 + (int)(21*Math.random());
    StringBuilder str = new StringBuilder();
    for (int i = 0; i < length; i++) {
        char ch = (char)('A' + (int)(26*Math.random()));
// a random letter
        str.append(ch);
    }
    return str.toString();
}
```

This method is then used to fill an array with random strings. I wrote methods for creating random arrays of a given size. I used `Arrays.copyOf()` to make copies of the arrays. I need two copies of each random array, since I want to apply `selectionSort()` and `Arrays.sort()` to identical data. So the arrays are created with

```
numberList1 = randomNumbers(SIZE);
numberList2 = Arrays.copyOf(numberList1, SIZE);
stringList1 = randomStrings(SIZE);
stringList2 = Arrays.copyOf(stringList1, SIZE);
```

where `SIZE` is a constant. (By using a constant here, I can easily adapt the program to run with different array sizes.)

The selection sort algorithm is copied from Subsection 7.5.4. We actually need two `selectionSort` methods, one to sort numbers an one to sort strings. You will see that my program includes tests of those methods. Without that test, I wouldn't have been confident that my subroutines were sorting the arrays correctly.

The code for timing the sorting algorithms is straightforward. Here are sort times from my program, running my computer:

```
Seconds to sort 100000 numbers with selectionSort: 3.39291
Seconds to sort 100000 numbers with Arrays.sort():
0.0369317
Seconds to sort 100000 strings with selectionSort: 31.9331
Seconds to sort 100000 strings with Arrays.sort():
0.0571503
```

Note that `Arrays.sort()` is **much** faster than `selectionSort`. Also, it takes longer to sort strings that it takes to sort numbers, since comparing two strings takes longer than comparing two numbers. The advantage of `Arrays.sort()` is greater for strings than it is for numbers. I'm not sure why that happens.

You can try running the program with different array sizes. A large difference is noticeable even for arrays of size 1000, although all of the times are rather short in that case. The advantage of `Arrays.sort()` increases as the array size increases.

The Solution

```
import java.util.Arrays;

public class SortExperiments {
```

```
    final static int SIZE = 100000; // The length of arrays
that will be sorted.

    /**
     * Creates a random string.  The length of the string
is between 5 and 25,
     * and it is made up of randomly selected uppercase
letters.
     */
    private static String randomString() {
        int length = 5 + (int)(21*Math.random());
        StringBuilder str = new StringBuilder();
        for (int i = 0; i < length; i++) {
            char ch = (char)('A' + (int)
(26*Math.random()));
            str.append(ch);
        }
        return str.toString();
    }

    /**
     * Creates an array of random real numbers.  The items
in the array
     * are random numbers in the range 0.0 to 1.0.
     * @param count The length of the array that is
created.
     */
    private static double[] randomNumbers(int count) {
        double[] numbers = new double[count];
        for (int i = 0; i < count; i++)
            numbers[i] = Math.random();
        return numbers;
    }

    /**
     * Creates an array of random strings. The items in the
     * array are created by calling the function
randomString();
     * @param count the size of the array that is created
     */
    private static String[] randomStrings(int count) {
        String[] strings = new String[count];
        for (int i = 0; i < count; i++)
            strings[i] = randomString();
        return strings;
    }
```

```
    /**
     * Sort an array of real numbers using the selection
sort algorithm.
     */
    private static void selectionSort(double[] numbers) {
        for (int top = numbers.length-1; top > 0; top-- ) {
            int maxloc = 0;
            for (int i = 1; i <= top; i++) {
                if (numbers[i] > numbers[maxloc])
                    maxloc = i;
            }
            double temp = numbers[top];
            numbers[top] = numbers[maxloc];
            numbers[maxloc] = temp;
        }
    }

    /**
     * Sort an array of strings using the selection sort
algorithm.
     */
    private static void selectionSort(String[] numbers) {
        for (int top = numbers.length-1; top > 0; top-- ) {
            int maxloc = 0;
            for (int i = 1; i <= top; i++) {
                if (numbers[i].compareTo(numbers[maxloc]) >
0)
                    maxloc = i;
            }
            String temp = numbers[top];
            numbers[top] = numbers[maxloc];
            numbers[maxloc] = temp;
        }
    }

    public static void main(String[] args) {

        long startTime;  // time when a sort begin.
        long endTime;    // time when a sort ends.

        double[] numberList1;  // An array of random
numbers.
        double[] numberList2;  // A copy of numberList1.

        String[] stringList1;  // An array of random
strings.
        String[] stringList2;  // A copy of stringList1.
```

```
        /* Make sure the selection sort methods are
correct.  The outputs
           should be correctly sorted. */

        System.out.println("First, test that selection sort
works on doubles.");
        System.out.println("The 10 output numbers should be
in increasing order.");
        numberList1 = randomNumbers(10);
        selectionSort(numberList1);
        for (double n : numberList1)
            System.out.println( "   " + n );
        System.out.println();

        System.out.println("Next, test that selection sort
works on strings.");
        System.out.println("The 10 output strings should be
in alphabetical order.");
        System.out.println("(Also tests that random strings
are made correctly.");
        stringList1 = randomStrings(10);
        selectionSort(stringList1);
        for (String str : stringList1)
            System.out.println( "   " + str );
        System.out.println();

        System.out.println();
        System.out.println("Times for sorting arrays of
size " + SIZE + ":");
        System.out.println();

        /* Create the arrays. */

        numberList1 = randomNumbers(SIZE);
        numberList2 = Arrays.copyOf(numberList1, SIZE);
        stringList1 = randomStrings(SIZE);
        stringList2 = Arrays.copyOf(stringList1, SIZE);

        /* Do the sorts and output the times. */

        startTime = System.nanoTime();
        selectionSort(numberList1);
        endTime = System.nanoTime();
        System.out.printf("Seconds to sort %d numbers with
selectionSort: %1.6g%n",
                                SIZE, (endTime - startTime)
/ 1e9);
```

```
        startTime = System.nanoTime();
        Arrays.sort(numberList2);
        endTime = System.nanoTime();
        System.out.printf("Seconds to sort %d numbers with
Arrays.sort(): %1.6g%n",
                                SIZE, (endTime - startTime)
/ 1e9);

        startTime = System.nanoTime();
        selectionSort(stringList1);
        endTime = System.nanoTime();
        System.out.printf("Seconds to sort %d strings with
selectionSort: %1.6g%n",
                                SIZE, (endTime - startTime)
/ 1e9);

        startTime = System.nanoTime();
        Arrays.sort(stringList2);
        endTime = System.nanoTime();
        System.out.printf("Seconds to sort %d strings with
Arrays.sort(): %1.6g%n",
                                SIZE, (endTime - startTime)
/ 1e9);
        System.out.println();

    }

}
```

Solution for Programming Exercise 7.4

Exercise 7.4:

In Exercise 6.2, you wrote a program `DragTwoSquares` that allows the user to drag a red square and a blue square around on a canvas. Write a much improved version where the user can add squares to a canvas and drag them around. In particular: If the user shift-clicks or right-clicks the canvas, then the user is trying to drag a square; find the square that contains the mouse position, if any, and move it as the user drags the mouse. Other clicks should add squares. You can place the center of the new square at the current mouse position. To make the picture more visually appealing, give each square a random color, and when you draw the squares, draw a black outline around each square. (My program also gives the square a random alpha value between 0.5 and 1.0).

Write a class to represent the data needed for drawing one square, and use an *ArrayList* to store the data for all the squares in the picture. If the user drags a square completely off the canvas, delete it from the list.

Discussion

We need three pieces of information in order to draw a square: its x-coordinate, its y-coordinate, and its color. We can pack the three pieces of information into a class:

```
/**
 * A class to hold to hold the data for one square.
 */
private static class SquareData {
    double x,y;  // The coordinates of the center of the
square.
    Color color; // The color of the square.
}
```

The data for the whole collection of squares is then stored using an instance variable named `squares` of type `ArrayList<SquareData>`. By using an ArrayList, we allow for an unlimited number of squares. (Note that since the xy coordinates will change when a square is dragged, we cannot use a record class here.)

Given this data structure, it's easy to write a `draw()` method that completely redraws the contents of the canvas. It can use a for-each loop to go through the *ArrayList*. The loop control variable, `squareData`, is of type *SquareData*, and the data needed to draw the square is given by its three instance variables `squareData.x`, `squareData.y`, and `squareData.color`. To draw a rectangle, we need the coordinates of the top-left corner of the square. Since `squareData.x` and `squareData.y` are the coordinates of the center of the square, we have to subtract half of the square size from them to get the coordinates for the top-left corner. In my program, the square size is 100, so the drawing method can be written as follows:

```
private void draw() {
    GraphicsContext g = canvas.getGraphicsContext2D();
    g.setFill(Color.rgb(230,255,230)); // light green
background
    g.fillRect(0,0,canvas.getWidth(),canvas.getHeight());
    g.setLineWidth(2);
    g.setStroke(Color.BLACK);
    for ( SquareData squareData: squares ) {
        g.setFill( squareData.color );
        g.fillRect( squareData.x - 50, squareData.y - 50,
100, 100 );
        g.strokeRect( squareData.x - 50, squareData.y -
50, 100, 100 );
    }
}
```

This method is called every time the mouse is moved while the user is dragging a square. It's not sufficient to draw the square in its new position—it must be removed from its previous position, and the area that it previously occupied has to be redrawn because it might contain other squares. It is easiest, though not most efficient, just to redraw the entire picture. When the user adds a new square, it would be enough to just draw the new square, but in fact my program just calls `draw()` in that case as well.

Another interesting point is in the `mousePressed()` routine, which is called when the user clicks the mouse on the drawing area. If the user shift-clicks or right-clicks, we have to figure out which square contains the mouse position, so that we know which square to drag. To do that, we have to search the *ArrayList* of squares to find one that meets the criterion that it contains the mouse position. (This is an example of linear search, as defined in Subsection 7.5.1.) There is a twist, since if there are two squares under the mouse, we want the one that is on top, and that will be the one that comes later in the array. So, we should search the list starting at the end and working towards the beginning. Then the first square that we encounter that contains the mouse position is the one that we want:

```
x = evt.getX();  // (x,y) is the point where the mouse was
pressed
y = evt.getY();
for (int i = squares.size() - 1; i >= 0; i--) {
    SquareData squareData = squares.get(i);
    double cx = squareData.x; // (cx,cy) is the center of
the square
    double cy = squareData.y;
    if ( x >= cx - 50 && x <= cx + 50 && y >= cy - 50 && y
<= cy + 50) {
        dragging = true;
        draggedSquare = squareData;
        offsetX = x - cx;
        offsetY = y - cy;
        break;  // stop as soon as we find  square
containing (x,y)
    }
}
```

You can check out the sample solution below for other details.

The Solution

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
```

```
import javafx.scene.input.MouseEvent;
import javafx.scene.input.MouseButton;

import java.util.ArrayList;


/**
 * A program that lets the user add squares to a canvas by
clicking.
 * The center of a square is placed at the point where the
user clicked.
 * Squares all have the same size (100-by-100).  They have
random
 * colors with up to 50% transparency.  If the user shift-
clicks
 * or right-clicks a square, the user can drag the square.
If
 * the user drags a square off the canvas, it is deleted
from the
 * list of squares.
 */
public class DragLotsOfSquares extends Application {

    public static void main(String[] args) {
        launch(args);
    }

    //-------------------------------------------------------
----------------

    /**
     * An object of type SquareData contains the data
necessary to
     * draw one square, that is, the color of the square
and the
     * coordinates of its center.
     */
    private static class SquareData {
        double x,y;  // Location of center of square.  The
size is always 100-by-100.
        Color color; // The color of the square
    }


    private ArrayList<SquareData> squares;  // Info for all
squares in the picture.

    private Canvas canvas;  // The canvas where the sqaures
```

are drawn.

```
    /**
     *  The start method sets up the GUI.  It adds mouse
event handlers to
     *  the canvas to implement adding and dragging
squares.
     */
    public void start(Stage stage) {

        squares = new ArrayList<SquareData>();

        canvas = new Canvas(640,480);
        draw(); // Will just fill canvas with background
color.

        canvas.setOnMousePressed( e -> mousePressed(e) );
        canvas.setOnMouseDragged( e -> mouseDragged(e) );
        canvas.setOnMouseReleased( e -> mouseReleased(e) );

        Pane root = new Pane(canvas);

        Scene scene = new Scene(root);

        stage.setScene(scene);
        stage.setTitle("Click to add a square. Right-click
to drag.");
        stage.setResizable(false);
        stage.show();
    }


    /**
     * Draw the canvas, showing all squares in their
current positions.
     */
    private void draw() {
        GraphicsContext g = canvas.getGraphicsContext2D();
        g.setFill(Color.rgb(230,255,230)); // light green

g.fillRect(0,0,canvas.getWidth(),canvas.getHeight());
        g.setLineWidth(2);
        g.setStroke(Color.BLACK);
        for ( SquareData squareData: squares ) {
            g.setFill( squareData.color );
            g.fillRect( squareData.x - 50, squareData.y -
50, 100, 100);
```

```
            g.strokeRect( squareData.x - 50, squareData.y -
50, 100, 100);
        }
    }


    //-----------------  Variables and methods for
responding to drags -----------

    private boolean dragging;      // Set to true when a
drag is in progress.

    private SquareData draggedSquare;  // When a drag is in
progress, this is
                                       // the square that
is being dragged.

    private double offsetX, offsetY;  // Offset of mouse-
click coordinates from the
                                      //   center of the
square that is being dragged.

    /**
     * Respond when the user presses the mouse on the
canvas.
     * A shift-click or right-click starts dragging the
square
     * under the mouse, if any.  Other clicks will add a
new
     * square with its center at the mouse position.  A
drag
     * operation is begun only if the user shift-clicks or
     * right-clicks a square.
     */
    public void mousePressed(MouseEvent evt) {

        if (dragging)  // Exit if a drag is already in
progress.
            return;

        double x = evt.getX();  // Location where user
clicked.
        double y = evt.getY();

        if (evt.isShiftDown() || evt.getButton() ==
MouseButton.SECONDARY) {
            // If user shift-clicked a square, start
dragging it.
```

```
            /* Find the square, if any, that contains
(x,y).  If several squares
             * contain (x,y), we want the one on top, which
is the LAST one in
             * the list that contains (x,y) -- so consider
the squares in the
             * reverse of their order in the list. */

            for (int i = squares.size() - 1; i >= 0; i--) {
                SquareData squareData = squares.get(i);
                double cx = squareData.x; // (cx,cy) is the
center of the square
                double cy = squareData.y;
                if ( x >= cx - 50 && x <= cx + 50 && y >=
cy - 50 && y <= cy + 50) {
                    dragging = true;
                    draggedSquare = squareData;
                    offsetX = x - cx;
                    offsetY = y - cy;
                    break;  // stop as soon as we find
square containing (x,y)
                }
            }
        }
        else { // Add a new square with center at (x,y)
            SquareData squareData = new SquareData();
            squareData.x = x;
            squareData.y = y;
                // Use a random color for the square.
Choose random RGB components
                // in the range 0.0 to 1.0 and a random
alpha in the range 0.5 to 1.0.
                // This will make a square that can be
somewhat transparent but not
                // too transparent.
            squareData.color = Color.color(
                    Math.random(), Math.random(),
Math.random(), 0.5 + 0.5*Math.random() );
            squares.add( squareData );
            draw();  // Redraw the whole picture to show
the new square.
                     //  (Could have just drawn it here
instead!)
        }
    }
```

```
    /**
     * Dragging stops when user releases the mouse button.
If the user
     * has dragged the square completely off the canvas,
then it is deleted
     * from the list of squares. (That will have no visible
effect on the
     * picture, so the canvas is not redrawn.)
     */
    public void mouseReleased(MouseEvent evt) {
        if ( ! dragging )
            return;

        /* To test if the square has moved completely off
the canvas,
         * test that its center is at least 50 pixels
outside the canvas. */

        if (draggedSquare.x > canvas.getWidth() + 50
                || draggedSquare.x < -50
                || draggedSquare.y > canvas.getHeight() +
50
                || draggedSquare.y < -50) {
              // Square is completely off the canvas, so
remove it!
            squares.remove(draggedSquare);
              // For testing, to make sure square is
actually deleted:
            System.out.println("Removed square; list size =
" + squares.size());
        }

        dragging = false;  // drag operation has ended.
        draggedSquare = null;
    }


    /**
     * Respond when the user drags the mouse.  If a square
is
     * not being dragged, then exit. Otherwise, change the
position
     * of the square that is being dragged to match the
position
     * of the mouse.  Note that the center of the square is
placed
     * in the same relative position with respect to the
mouse that it
```

```
     * had when the user started dragging it.
     */
    public void mouseDragged(MouseEvent evt) {
        if ( ! dragging )
            return;
        double x = evt.getX();
        double y = evt.getY();
        draggedSquare.x = x - offsetX;
        draggedSquare.y = y - offsetY;
        draw();  // Redraw picture to show square in new
positions.
    }

} // end class DragLotsOfSquares
```

Solution for Programming Exercise 7.5

Exercise 7.5:

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. Do **not** use any built-in function such as `Arrays.sort()`. Do the sorting yourself.

Discussion

The sample program ReverseWithDynamicArray.java from Subsection 7.2.4 reads in up to 100 positive integers from the user and outputs them in the reverse of the order in which the user entered them. This is similar to what we have to do for this exercise, except that the numbers we have to read are real numbers (of type double) and they have to be output in sorted order.

There are two basic approaches to this problem. The first is to store all the numbers in an array in the order in which they are input. After all the numbers have been input, the array can be sorted, and then the contents of the array can be output. The second approach is to always keep the array in sorted order as numbers are added to it. When a new number is input, that number must be inserted into its correct location in the array, in order to keep the array sorted. After all the numbers have been input, the contents of the array are ready to be printed.

Two solutions to the exercise, based on these two approaches, are shown below. They use techniques for sorting and inserting that were covered in Section 7.5. In my first program, I've chosen to use Selection Sort to sort the array. Insertion Sort would work just as well. The Selection Sort subroutine is taken from Section 7.5 with two changes: It sorts an array of double values instead of an array of `ints`, and it has been modified to work with a

"partially full" array. In order to make the subroutine work with a partially full array, it is necessary to add a parameter that tells the subroutine how many entries in the array are in use. The modified Selection Sort routine is as follows, with changes from the original version shown in red italic

```
static void selectionSort(double[] A, int count) {
      // Sort the numbers in A[0], A[1], ..., A[count-1]
into
      // increasing order using Selection Sort.
   for ( int lastPlace = count - 1; lastPlace > 0;
lastPlace-- ) {
      int maxLoc = 0;
      for (int j = 1; j <= lastPlace; j++) {
         if (A[j] > A[maxLoc]) {
            maxLoc = j;
         }
      }
      double temp = A[maxLoc];
      A[maxLoc] = A[lastPlace];
      A[lastPlace] = temp;
   }
} // end selectionSort
```

In the first version of the program, this subroutine is called just after all the numbers have been input from the user.

The second version of the program is straightforward. It uses the `insert()` subroutine from Section 7.5, modified to work with an array of `doubles` instead of an array of `ints`.

The Solution

First solution, with Selection Sort:

```
import textio.TextIO;

    /**
     * This program reads up to 100 positive integers from
the user and
     * prints them in sorted order.  Input ends when the
user enters a
     * non-positive integer.  The numbers are read and
```

```
stored in an array.
     * That array is sorted using selection sort, and then
the array is
     * printed.
     */

    public class SortInputNumbers {

       public static void main(String[] args) {

          double[] numbers;  // An array for storing the
input values.
          int numCt;         // The number of numbers saved
in the array.
          double num;        // One of the numbers input by
the user.

          numbers = new double[100];   // Space for 100
numbers.
          numCt = 0;                   // No numbers have
been saved yet.

          System.out.println("Enter up to 100 positive
numbers; Enter 0 to end");

          while (true) {   // Get the numbers and put them
in the array.
             System.out.print("? ");
             num = TextIO.getlnDouble();
             if (num <= 0)
                break;
             numbers[numCt] = num;
             numCt++;
          }

          selectionSort(numbers, numCt);  // Sort the
numbers.

          System.out.println("\nYour numbers in sorted
order are:\n");

          for (int i = 0; i < numCt; i++) {
              System.out.println( numbers[i] );
          }

       } // end main();

       /**
```

```
        * Sort the numbers in A[0], A[1], ..., A[count-1]
into
        * increasing order using Selection Sort.
        */
       static void selectionSort(double[] A, int count) {
          for ( int lastPlace = count - 1; lastPlace > 0;
lastPlace-- ) {
             int maxLoc = 0;
             for (int j = 1; j <= lastPlace; j++) {
                if (A[j] > A[maxLoc]) {
                   maxLoc = j;
                }
             }
             double temp = A[maxLoc];
             A[maxLoc] = A[lastPlace];
             A[lastPlace] = temp;
          }
       } // end selectionSort

    }  // end class SortInputNumbers
```

Second solution, with Insert:

```
import textio.TextIO;

/**
 * This program reads up to 100 positive integers from the
user and
 * prints them in sorted order.  Input ends when the user
enters a
 * non-positive integer.  The numbers are read and inserted
into
 * an array.  The array is maintained at all times in
sorted order.
 */

public class SortInputNumbers2 {

   public static void main(String[] args) {

      double[] numbers;  // An array for storing the input
values.
      int numCt;         // The number of numbers saved in
the array.
      double num;        // One of the numbers input by the
user.
```

```
      numbers = new double[100];   // Space for 100
numbers.
      numCt = 0;                   // No numbers have been
saved yet.

      System.out.println("Enter up to 100 positive numbers;
Enter 0 to end");

      while (true) {   // Get the numbers and insert them
into the array.
         System.out.print("? ");
         num = TextIO.getlnDouble();
         if (num <= 0)
            break;
         insert(numbers, numCt, num);
         numCt++;
      }

      System.out.println("\nYour numbers in sorted order
are:\n");

      for (int i = 0; i < numCt; i++) {
          System.out.println( numbers[i] );
      }

   } // end main();

   /**
    * Assume that A contains itemsInArray in increasing
order.
    * Insert newItem into its correct position in the
sorted array.
    */
   static void insert(double[] A, int itemsInArray, double
newItem) {
      int loc = itemsInArray - 1;
      while (loc >= 0 && A[loc] > newItem) {
             // Move the item from A[loc] up one space.
         A[loc + 1] = A[loc];
         loc = loc - 1;
      }
      A[loc + 1] = newItem;  // Put newItem in the last
vacated space.
   } // end insert

}  // end class SortInputNumbers2
```

Solution for Programming Exercise 7.6

Exercise 7.6:

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 2.4.4, you can use *TextIO* to read and write files. Use a variable of type `ArrayList<String>` to store the words. It is not easy to separate a file into words as you are reading it, especially if you want to allow apostrophes in the middle of a word. You can use the following method in your program:

```
/**
 * 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.

Discussion

This is actually not a very difficult program to write. The main point of the exercise is to get you to use a list of strings and to do something with files.

There are several possible approaches to this problem. One approach is to simply dump all the words from the file into a list, without worrying about eliminating duplicates or keeping the list in order. After the file has been read, the list can be sorted and printed. Although the list can contain duplicates, the output file should list each word only once. However, it's easy to leave out the duplicates as the file is being written. If `wordList` is the variable of type `ArrayList<String>` that holds the already sorted list of words, then the following code will output the list without duplicates. The idea is that a word is written only if it is different from the previous word in the list. Word number 0 is a special case, because there is no previous location in the list in that case:

```
for (int i = 0; i < wordlist.size(); i++) {
   if (i == 0 || ! wordlist.get(i).equals(wordList.get(i-
1) )
      System.out.println(wordlist.get(i));
}
```

This is actually the most efficient of the three solutions that I discuss, as long as the sorting is done with an efficient method. (For arrays, sorting can be done using `Arrays.sort()`. I haven't discussed a similar method for *ArrayLists*, but an `ArrayList<String>` can be sorted using a static method `Collections.sort(list)` from the class `java.util.Collections`.)

A second approach to the problem would be to keep the list of words in sorted order as it is being constructed. This can be done by applying the `insert()` routine from Subsection 7.5.3 to insert each word into the list.

A third approach, and the one that I use in my solution, is to eliminate duplicates from the list as it being constructed. Each time a word is read from the input file, I first check whether the word is already in the list. If so, I discard it; if not, I add it to the end of the list. The function `wordList.indexOf(word)` can be used to test whether a given word is already in the list; this function returns the value `-1` if `word` is not in the list. After the input file has been read, the list contains one copy of each word that was found in the file. At this point the list still has to be sorted. I use a

selection sort algorithm (Subsection 7.5.4), adapted to work for an *ArrayList*, to do the sorting. Then, all the elements of the list are output using a for-each loop:

```
for (String w : wordList)
   TextIO.putln("   " + w);
```

To let the user select the input and the output files, I use the methods `TextIO.readUserSelectedFile()` and `TextIO.writeUserSelectedFile()`, which are discussed in Subsection 2.4.4. These methods put up a file dialog where the user can select a file. After the user selects an input file, `TextIO` reads from that file instead of from the user's input. After the user selects an output file, `TextIO` writes to that file instead of to standard output. If the user cancels the input file dialog, then there is no input file to process, so I exit the program. If the user cancels the output file dialog, I write the list of words anyway—it will go to standard output so the user will see it on the screen.

When `TextIO` is working with files and an error occurs, it will generate an error of type *IllegalArgumentException*. My program catches the error if one occurs, and it prints an error message. An error is not very likely in this case but one could occur if, for example, the user selects an input file that the user does not have permission to read.

The complete code for my solution is shown below.

The Solution

```
import java.util.ArrayList;
import textio.TextIO;

/**
 * Makes an alphabetical list of all the words in a file
selected
 * by the user.  The list can be written to a file.
 */
public class ListAllWordsFromFile {


   public static void main(String[] args) {
```

```
      System.out.println("""
                  This program will ask you to select an
input file
              It will read that file and make an
alphabetical
              list of all the words in the file.  After
reading
              the file, the program asks you to select an
output
              file.  If you select a file, the list of
words will
              be written to that file; if you cancel, the
list
              be written to standard output.  All words are
converted
              to lower case, and duplicates are eliminated
from the list.

              Press return to begin.

              """);

      TextIO.getln();  // Wait for user to press return.

      try {
         if (TextIO.readUserSelectedFile() == false) {
            System.out.println("No input file selected.
Exiting.");
             System.exit(1);
         }
         ArrayList<String> wordList = new ArrayList<String>
();
         String word = readNextWord();
         while (word != null) {
            word = word.toLowerCase();  // convert word to
lower case
            if ( wordList.indexOf(word) == -1 ) {
                  // This is a new word, so add it to the
list
               wordList.add(word);
            }
            word = readNextWord();
         }
         System.out.println("Number of different words
found in file:  "
               + wordList.size());
         System.out.println();
```

```
         if (wordList.size() == 0) {
            System.out.println("No words found in file.");
            System.out.println("Exiting without saving
data.");
            System.exit(0);
         }
         selectionSort(wordList);
         TextIO.writeUserSelectedFile(); // If user
cancels, output automatically
                                         // goes to
standard output.
         TextIO.putln(wordList.size() + " words found in
file:\n");
         for (String w : wordList)
            TextIO.putln("   " + w);
         System.out.println("\n\nDone.\n\n");
      }
      catch (Exception e) {
         System.out.println("Sorry, an error has
occurred.");
         System.out.println("Error Message:  " +
e.getMessage());
      }
      System.exit(0);  // Might be necessary, because of
use of file dialogs.
   }


   /**
    * Sorts a list of strings into lexicographical order,
using
    * selection sort and treating the list much like an
array.  In this
    * program, the list only contains words made up of
lower case
    * letters, so lexicographic order is the same as
alphabetical order.
    */
   private static void selectionSort(ArrayList<String>
list) {
      for (int top = list.size() - 1; top > 0; top--) {
         int indexOfBiggest = 0;
         for (int j = 0; j < top; j++) {
            String str = list.get(j);
            if (str.compareTo( list.get(indexOfBiggest) ) >
0) {
               indexOfBiggest = j;
            }
```

```
         }
         String temp = list.get(top);
         list.set( top, list.get(indexOfBiggest) );
         list.set( indexOfBiggest, temp );
      }
   }


   /**
    * 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 that the next character, 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.
   }

} // end ListAllWordsFromFile
```

Solution for Programming Exercise 7.7

Exercise 7.7:

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. The board should have 13 rows and 13 columns of squares. 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 7.6.3. Play alternates strictly between the two players, and there is no need to highlight the legal moves. You will only need one nested subclass, a subclass of *Canvas* to draw the board and do all the work of the game, like the nested *CheckersBoard* class in the *Checkers* program. 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 program. 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 player's 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, just after black has won the game.

Discussion

This is a fairly complicated program, but it's possible to design and build it in stages, testing each stage separately. The first stage, the general layout of the panel, is already done in the Checkers.java program. With just a few changes, the main canvas class, the layout of the board, and the button and message handling come directly from that program. Let's take the rest of the Go Moku game one stage at a time.

A two-dimensional array is used to store the contents of the board. This array is of type `int[][]` and is named `board`. It is defined as an instance variable in the *GoMokuBoard* class, and it is initialized in the constructor of that class to be a 13-by-13 array. The value in each position of the array is one of three constants: `EMPTY`, `WHITE`, or `BLACK`. When a game begins, each of the entries in the array is set to empty. When a player clicks on an empty square, the corresponding entry in the array is changed from `EMPTY` to `BLACK` or `WHITE`, depending on which player placed the piece. In the `drawBoard()` method, the contents of the `board` array are used to decide what pieces to draw on the board.

Drawing the Board

We need a `drawBoard()` method for the board class that can draw the board. The board has 13 rows and 13 columns of spaces. How wide should the board be? If each square in the board is `x` pixels wide, we need a total of `13*x` pixels just for the spaces. But there are also lines between the spaces and at the edges of the board. I make each line two pixels wide, so the 14 lines require another 28 pixels. So, with squares of side `x`, we need a board that is `13*x+28` pixels wide. Since I wanted something about the same size as the original checkerboard, which was 324 pixels, I choose `x` to be 22, giving a board width of 314 pixels. The height is also 314.

The left edge of the `col`-th column of squares in the board is `2+24*col`. This allows for the two-pixel border on the left and for 24 pixels for each of the preceding columns of squares. (That's 22 pixels for the square plus two pixels for the line between that column and the next.) The lines at `x` values `1 + 24*i`, for `i` from 0 to 13. This puts the center of the pen that is used to draw the line in the correct position, one pixel to the left of each column for the first 13 lines, and one pixel from the right edge of the canvas for the final

line. Rows work the same way. To draw a piece in row number `row` and column number `col`, the command

```
g.fillOval(4 + 24*col, 4 + 24*row, 18, 18);
```

can be used. This allows a two-pixel gap between the oval that represents that piece and the side of the square. Since the size of the square is 22 and there is a 2-pixel gap on each side of the oval, the size of the oval is 22 minus 4, or 18. In my program, I defined a `drawPiece()` method to draw a piece.

The `drawBoard()` method fills the canvas with light gray. It then draws the black lines between the squares, and draws all the pieces that have been placed on the board. Remember that the piece in a given row and column is recorded in the `board` array as `board[row][col]`. This is done as follows:

```
/* Draw lines separating the square and along the edges of
the canvas.  */

 g.setStroke(Color.BLACK);
 for (int i = 0; i <= 13; i++) {
     g.strokeLine(1 + 24*i, 0, 1 + 24*i, 314);
     g.strokeLine(0, 1 + 24*i, 314, 1 + 24*i);
 }

 /* Draw the pieces that are on the board. */

 for (int row = 0; row < 13; row++)
     for (int col = 0; col < 13; col++)
         if (boardData[row][col] != EMPTY)
             drawPiece(g, boardData[row][col], row, col);
```

Playing the Game

The logic of the GoMoku game itself is mostly in the method "`void doClickSquare(int row, int col)`", which is called by the `mousePressed()` method when the user clicks on the square in row number `row` and column number `col`. This method must check whether the move is legal. If so, the move is made. The method then checks whether the move

wins the game. If so, the game ends. The game will also end if the board has become completely full. Otherwise, play passes to the other player.

The current player is recorded in an instance variable named `currentPlayer`. The value of this variable is one of the two constants `WHITE` or `BLACK`. The game can be ended by calling a method named `gameOver()`. I wrote a boolean-valued method called `winner()` to check whether a move wins the game. (When I first wrote this method, it did nothing but "`return false`". This let me try out the program at this stage of development, before I started working on the difficult problem of testing for a winner.) The `doClickSquare()` method can be written:

```
void doClickSquare(int row, int col) {

    /* Check that the user clicked an empty square.  If
not, show an
      error message and exit. */

    if ( boardData[row][col] != EMPTY ) {
        if (currentPlayer == BLACK)
            message.setText("BLACK:  Please click an empty
square.");
        else
            message.setText("WHITE:  Please click an empty
square.");
        return;
    }

    /* Make the move.  Check if the board is full or if
the move
      is a winning move.  If so, the game ends.  If not,
then it's
      the other user's turn.  If there is a winner, call
      drawWinLine() to mark the winning pieces. */

    boardData[row][col] = currentPlayer;  // Make the
move.
    drawBoard();

    if (winner(row,col)) {  // First, check for a winner.
        if (currentPlayer == WHITE)
            gameOver("WHITE wins the game!");
        else
            gameOver("BLACK wins the game!");
        drawWinLine();
```

```
        return;
    }

    boolean emptySpace = false;     // Check if the board
is full.
    for (int i = 0; i < 13; i++)
        for (int j = 0; j < 13; j++)
            if (boardData[i][j] == EMPTY)
                emptySpace = true;
    if (emptySpace == false) {
        gameOver("The game ends in a draw.");
        return;
    }

    /* Continue the game.  It's the other player's turn.
*/

    if (currentPlayer == BLACK) {
        currentPlayer = WHITE;
        message.setText("WHITE:  Make your move.");
    }
    else {
        currentPlayer = BLACK;
        message.setText("BLACK:  Make your move.");
    }

}  // end doClickSquare()
```

Note that when the move wins the game, the method `drawWinLine()` is called. This will draw a red line through the winning pieces. Unlike every other part of the board, the red line is not drawn by `drawBoard()`. It is added when the game is over, and it will stay there until the board is erased at the beginning of the new game. Note that this violates my general policy that a `draw()` method should completely reflect the state of the game. In this case, it does not draw the red line when the state of the game is that the game has been won. This is OK only because `drawBoard()` is never called when the game is in that state.

Determining the Winner

The `winner()` method is certainly the hardest part of the program. The method must look in each of the four possible directions from the square

where the user has placed a piece. If the player has five or more pieces in a row in that direction, then the player has won. As indicated in the exercise, a direction can be indicated by two variables, `dirX` and `dirY`. The values of these variables for each of the four directions are:

```
                      dirX    dirY    Why?
                      ----    ----    --------------------
------------
horizontal direction    1       0       Only x changes.
vertical direction      0       1       Only y changes.
falling diagonal        1       1       Both x and y
change.
rising diagonal         1      -1       Change in opposing
directions.
```

I wrote a method "`int count(int player, int row, int col, int dirX, int dirY)` that counts the number of pieces the specified player has in a row, starting from the square in row number `row` and column number `col` and looking in the direction indicated by `dirX` and `dirY`. This method contains the code given in the exercise. It returns the number of pieces found. My `winner` method just calls this method for each of the four directions:

```
/**
 * This is called just after a piece has been played on
the
 * square in the specified row and column.  It determines
 * whether that was a winning move by counting the number
 * of squares in a line in each of the four possible
 * directions from (row,col).  If there are 5 squares (or
more)
 * in a row in any direction, then the game is won.
 */
private boolean winner(int row, int col) {

   if (count( board[row][col], row, col, 1, 0 ) >= 5)
      return true;
   if (count( board[row][col], row, col, 0, 1 ) >= 5)
      return true;
   if (count( board[row][col], row, col, 1, -1 ) >= 5)
      return true;
   if (count( board[row][col], row, col, 1, 1 ) >= 5)
      return true;

   /* When we get to this point, we know that the game is
```

```
not won. */

   return false;

}  // end winner()
```

When I first wrote this method, I checked whether the number of pieces was "`== 5`" instead of "`>= 5`". This was a bug. It's possible for a player to get more than 5 pieces in a row, if the player plays a piece in an empty square that joins two shorter rows of pieces together.

Marking the Winning Pieces

In my program, when a player wins, the row of pieces that wins the game is marked with a red line. To do this, I added four instance variables of type int to the *GoMokuBoard* class. The instance variables are named `win_r1`, `win_c1`, `win_r2`, and `win_c2`. After a player has won the game, the values of these variables have been set to mark the squares at the two ends of the winning row of pieces. The positions of these squares are given by `(win_r1,win_c1)` and `(win_r2,win_c2)`.

I added some code to the `count()` method to set the values of these variables. As the `count()` method counts the pieces in a line, it sets `win_r1`, `win_c1`, `win_r2`, and `win_c2` to mark the location of the last piece it finds in the two directions it checks. The values are always set when `count()` is called. If the game is won, this will set the values correctly. My program depends on the fact that once it has been determined that a game has been won, `count()` will not be called again during that game. This means that when the red line is drawn, the variables `win_r1`, `win_c1`, `win_r2`, and `win_c2` will still mark the ends of the line of pieces that won the game. (I'm not really happy about how tricky this is. My solution is "fragile" in the sense that it could easily be broken inadvertently if the logic of the program is changed just a little. But I really wanted to mark the winning pieces...)

The Solution

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.layout.Pane;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.input.MouseEvent;

/**
 * This panel lets two users play Go Moku against each
other.
 * Black always starts the game.
 */
public class GoMoku extends Application {

    public static void main(String[] args) {
        launch(args);
    }

    //-------------------------------------------------------
----------------

    private static final int  // Constants to represent
possible contents
             EMPTY = 0,       //     of squares on the
board.  The constants
             BLACK = 1,       //     BLACK and WHITE are
also used to
             WHITE = 2;       //     represent the current
player.

    private GoMokuBoard board; // A canvas on which a
checker board is drawn,
                           // defined by a static nested
subclass.  Much of
                           // the game logic is defined in
this class.


    private Button newGameButton;  // Button for starting a
new game.
```

```
    private Button resignButton;   // Button that a player
can use to end
                                    // the game by
resigning.

    private Label message;  // Label for displaying
messages to the user.

    /**
     * The constructor creates the Board (which in turn
creates and manages
     * the buttons and message label), adds all the
components, and sets
     * the bounds of the components.  A null layout is
used.  (This is
     * the only thing that is done in the main Checkers
class.)
     */
    public void start(Stage stage) {

        /* Create the label that will show messages. */

        message = new Label("Click \"New Game\" to
begin.");
        message.setTextFill( Color.rgb(100,255,100) ); //
Light green.
        message.setFont( Font.font(null, FontWeight.BOLD,
18) );

        /* Create the buttons and the board.  The buttons
MUST be
         * created first, since they are used in the
CheckerBoard
         * constructor! */

        newGameButton = new Button("New Game");
        resignButton = new Button("Resign");

        board = new GoMokuBoard(); // a subclass of Canvas,
defined below
        board.drawBoard();  // draws the empty board

        /* Set up ActionEvent handlers for the buttons and
a MousePressed handler
         * for the board.  The handlers call instance
methods in the board object. */

        newGameButton.setOnAction( e -> board.doNewGame()
```

```
);
        resignButton.setOnAction( e -> board.doResign() );
        board.setOnMousePressed( e -> board.mousePressed(e)
);

        /* Set the location of each child by calling its
relocate() method */

        board.relocate(20,20);
        newGameButton.relocate(370, 120);
        resignButton.relocate(370, 200);
        message.relocate(20, 370);

        /* Set the sizes of the buttons.  For this to have
an effect, make
         * the butons "unmanaged."  If they are managed,
the Pane will set
         * their sizes. */

        resignButton.setManaged(false);
        resignButton.resize(100,30);
        newGameButton.setManaged(false);
        newGameButton.resize(100,30);

        /* Create the Pane and give it a preferred size.
If the
         * preferred size were not set, the unmanaged
buttons would
         * not be included in the Pane's computed preferred
size. */

        Pane root = new Pane();

        root.setPrefWidth(500);
        root.setPrefHeight(420);

        /* Add the child nodes to the Pane and set up the
rest of the GUI */

        root.getChildren().addAll(board, newGameButton,
resignButton, message);
        root.setStyle("-fx-background-color: darkgreen; "
                            + "-fx-border-color: darkred; -
fx-border-width:3");
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setResizable(false);
        stage.setTitle("Go Moku!");
```

```
        stage.show();

    } // end start()


    /**
     * This canvas displays a GoMoku board: a 314-by-314
board divided into 13
     * rows and 13 columns of squares. This class contains
methods that are
     * called in response to a mouse click on the canvas
and in response to
     * clicks on the New Game and Resign buttons.   Note
that the "New Game"
     * and "Resign" buttons must be created before the
Board constructor is
     * called, since the constructor references the buttons
(in the call
     * to doNewGame()).
     */
    private class GoMokuBoard extends Canvas {

        int[][] boardData;

        boolean gameInProgress; // Is a game currently in
progress?

        /* The next three variables are valid only when the
game is in progress. */

        int currentPlayer;      // Whose turn is it now?
The possible values
                                //    are BLACK and WHITE

        int win_r1, win_c1, win_r2, win_c2; // When a
player wins by getting five or more
                                            // pieces in a
row, the squares at the
                                            // ends of the
row are (win_r1,win_c1)
                                            // and
(win_r2,win_c2).  A red line is
                                            // drawn
between these squares.  The values are
                                            // set in the
count() method.  The are only used
                                            // in
drawWinLine(), which is only called after
```

```
                                                // the game has
been won.


        /**
         * Constructor.  Creates a CheckersData to
represent the
         * contents of the checkerboard, and calls
doNewGame to
         * start the first game.
         */
        GoMokuBoard() {
            super(314,314);  // canvas is 314-by-314 pixels
            doNewGame();
        }

        /**
         * Start a new game.  This method is called when
the Board is first
         * created and when the "New Game" button is
clicked.  Event handling
         * is set up in the start() method in the main
class.
         */
        void doNewGame() {
            if (gameInProgress == true) {
                    // This should not be possible, but it
doesn't hurt to check.
                message.setText("Finish the current game
first!");
                return;
            }
            boardData = new int[13][13];   // Start the
game with an empty board.
                                            //  This relies on
the fact that EMPTY = 0.
            currentPlayer = BLACK;   // BLACK moves first.
            message.setText("Black:  Make your move.");
            gameInProgress = true;
            newGameButton.setDisable(true);
            resignButton.setDisable(false);
            drawBoard();
        }

        /**
         * Current player resigns.  Game ends.  Opponent
wins.  This method is
         * called when the user clicks the "Resign" button.
```

```
Event handling is
         * set up in the start() method in the main class.
         */
        void doResign() {
            if (gameInProgress == false) {  // Should be
impossible.
                message.setText("There is no game in
progress!");
                return;
            }
            if (currentPlayer == WHITE)
                gameOver("WHITE resigns.  BLACK wins.");
            else
                gameOver("BLACK resigns.  WHITE wins.");
        }

        /**
         * The game ends.  The parameter, str, is displayed
as a message
         * to the user.  The states of the buttons are
adjusted so players
         * can start a new game.  This method is called
when the game
         * ends at any point in this class.
         */
        void gameOver(String str) {
            message.setText(str);
            newGameButton.setDisable(false);
            resignButton.setDisable(true);
            gameInProgress = false;
        }


        /**
         * This is called by mousePressed() when a player
clicks on the
         * square in the specified row and col.  It has
already been checked
         * that a game is, in fact, in progress.
         */
        void doClickSquare(int row, int col) {

            /* Check that the user clicked an empty square.
If not, show an
              error message and exit. */

            if ( boardData[row][col] != EMPTY ) {
                if (currentPlayer == BLACK)
```

```
                    message.setText("BLACK:  Please click
an empty square.");
                else
                    message.setText("WHITE:  Please click
an empty square.");
                return;
            }

            /* Make the move.  Check if the board is full
or if the move
              is a winning move.  If so, the game ends.  If
not, then it's
              the other user's turn.  If there is a winner,
call
              drawWinLine() to mark the winning pieces. */

            boardData[row][col] = currentPlayer;  // Make
the move.
            drawBoard();

            if (winner(row,col)) {  // First, check for a
winner.
                if (currentPlayer == WHITE)
                    gameOver("WHITE wins the game!");
                else
                    gameOver("BLACK wins the game!");
                drawWinLine();
                return;
            }

            boolean emptySpace = false;     // Check if the
board is full.
            for (int i = 0; i < 13; i++)
                for (int j = 0; j < 13; j++)
                    if (boardData[i][j] == EMPTY)
                        emptySpace = true;
            if (emptySpace == false) {
                gameOver("The game ends in a draw.");
                return;
            }

            /* Continue the game.  It's the other player's
turn. */

            if (currentPlayer == BLACK) {
                currentPlayer = WHITE;
                message.setText("WHITE:  Make your move.");
            }
```

```
            else {
                currentPlayer = BLACK;
                message.setText("BLACK:  Make your move.");
            }

        }  // end doClickSquare()


        /**
         * This is called just after a piece has been
played on the
         * square in the specified row and column.  It
determines
         * whether that was a winning move by counting the
number
         * of squares in a line in each of the four
possible
         * directions from (row,col).  If there are 5
squares (or more)
         * in a row in any direction, then the game is won.
The values
         * of the variables win_r1, win_c1, win_r2, win_c2
are set in
         * any case, but they are only valid -- and will
only be used --
         * if the game has actually been won.
         */
        private boolean winner(int row, int col) {

            if (count( boardData[row][col], row, col, 1, 0
) >= 5)
                return true;
            if (count( boardData[row][col], row, col, 0, 1
) >= 5)
                return true;
            if (count( boardData[row][col], row, col, 1, -1
) >= 5)
                return true;
            if (count( boardData[row][col], row, col, 1, 1
) >= 5)
                return true;

            /* When we get to this point, we know that the
game is not won.*/

            return false;

        }  // end winner()
```

```
        /**
         * Counts the number of the specified player's
pieces starting at
         * square (row,col) and extending along the
direction specified by
         * (dirX,dirY).  It is assumed that the player has
a piece at
         * (row,col).  This method looks at the squares
(row + dirX, col + dirY),
         * (row + 2*dirX, col + 2*dirY), ... until it hits
a square that is
         * off the board or is not occupied by one of the
player's pieces.
         * It counts the squares that are occupied by the
player's pieces.
         * Furthermore, it sets (win_r1,win_c1) to mark
last position where
         * it saw one of the player's pieces.  Then, it
looks in the
         * opposite direction, at squares (row - dirX, col-
dirY),
         * (row - 2*dirX, col - 2*dirY), ... and does the
same thing.
         * Except, this time it sets (win_r2,win_c2) to
mark the last piece.
         * Note:  The values of dirX and dirY must be 0, 1,
or -1.  At least
         * one of them must be non-zero.
         */
        private int count(int player, int row, int col, int
dirX, int dirY) {

            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 &&
boardData[r][c] == player ) {
                // Square is on the board and contains one
of the players's pieces.
                ct++;
                r += dirX;  // Go on to next square in this
```

```
direction.
                c += dirY;
            }

            win_r1 = r - dirX;  // The next-to-last square
looked at.
            win_c1 = c - dirY;  //    (The LAST one looked
at was off the board or
            //    did not contain one of the player's
pieces.

            r = row - dirX;  // Look in the opposite
direction.
            c = col - dirY;
            while ( r >= 0 && r < 13 && c >= 0 && c < 13 &&
boardData[r][c] == player ) {
                // Square is on the board and contains one
of the players's pieces.
                ct++;
                r -= dirX;   // Go on to next square in
this direction.
                c -= dirY;
            }

            win_r2 = r + dirX;
            win_c2 = c + dirY;

            // At this point, (win_r1,win_c1) and
(win_r2,win_c2) mark the endpoints
            // of the line of pieces belonging to the
player.

            return ct;

        }  // end count()


        /**
         * Draws the board and the pieces on the board.
This method does NOT
         * draw the red line through the winning pieces
after the game has
         * been won.
         */
        public void drawBoard() {

            GraphicsContext g = getGraphicsContext2D();
            g.setFill( Color.LIGHTGRAY );  // fill canvas
```

```
with light gray
            g.fillRect(0,0,314,314);

            /* Draw lines separating the square and along
the edges of the canvas.  */

            g.setStroke(Color.BLACK);
            g.setLineWidth(2);
            for (int i = 0; i <= 13; i++) {
                g.strokeLine(1 + 24*i, 0, 1 + 24*i, 314);
                g.strokeLine(0, 1 + 24*i, 314, 1 + 24*i);
            }

            /* Draw the pieces that are on the board. */

            for (int row = 0; row < 13; row++)
                for (int col = 0; col < 13; col++)
                    if (boardData[row][col] != EMPTY)
                        drawPiece(g, boardData[row][col],
row, col);

        }  // end drawBoard()


        /**
         * Draw a piece in the square at (row,col).  The
color is specified
         * by the piece parameter, which should be either
BLACK or WHITE.
         */
        private void drawPiece(GraphicsContext g, int
piece, int row, int col) {
            if (piece == WHITE) {
                g.setFill(Color.WHITE);
                g.fillOval(4 + 24*col, 4 + 24*row, 18, 18);
                g.setStroke(Color.BLACK);
                g.setLineWidth(1);
                g.strokeOval(4 + 24*col, 4 + 24*row, 18,
18);
            }
            else {
                g.setFill(Color.BLACK);
                g.fillOval(4 + 24*col, 4 + 24*row, 18, 18);
            }
        }


        /**
```

```
         * Draw a 4-pixel wide red line from the middle of
the square at
         * (win_r1,win_c1) to the middle of the square at
(win_r2,win_c2).
         * The values of the variables are set in the
count() method.
         * This routine is called to mark the pieces that
won the game.
         * It can only be called once in a given game, at
the point where
         * the game ends because a user places a winning
piece.
         */
        private void drawWinLine() {
            GraphicsContext g = getGraphicsContext2D();
            g.setStroke(Color.RED);
            g.setLineWidth(4);
            g.strokeLine( 13 + 24*win_c1, 13 + 24*win_r1,
13 + 24*win_c2, 13 + 24*win_r2 );
        }


        /**
         * Respond to a user click on the board.  If no
game is
         * in progress, show an error message.  Otherwise,
find
         * the row and column that the user clicked and
call
         * doClickSquare() to handle it.
         */
        public void mousePressed(MouseEvent evt) {
            if (gameInProgress == false)
                message.setText("Click \"New Game\" to
start a new game.");
            else {
                int col = (int)((evt.getX() - 2) / 24);
                int row = (int)((evt.getY() - 2) / 24);
                if (col >= 0 && col < 13 && row >= 0 && row
< 13)
                    doClickSquare(row,col);
            }
        }

    }  // end nested class GoMokuBoard


} // end class GoMoku
```

Solution for Programming Exercise 8.1

Exercise 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.

Discussion

This is really just a fairly easy exercise in using exceptions. The `root()` subroutine must be called in a `try..catch` statement. There must be a `catch` clause to handle the *IllegalArgumentException* that might be thrown by the routine. The catch clause can simply print an error message:

```
try {
   solution = root(A,B,C);
   System.out.println("A solution of the equation is " +
solution);
}
catch (IllegalArgumentException e) {
   System.out.println("Sorry, I can't find a solution.");
   System.out.println(e.getMessage());
}
```

Note that I've put the output statement that prints the solution inside the `try` statement. If an `IllegalArgumentException` is thrown by the subroutine, then this output statement will not be executed since the computer will jump immediately to the `catch` clause. You have to be careful about things like this. It wouldn't do to have the output statement after the `try..catch` statement, since then the computer would still try to execute the output statement after handling an *IllegalArgumentException*. There are other ways to deal with this problem. For example, since the `try..catch` statement occurs in a `while` loop, we could put a `continue` statement in the `catch` clause to abort further processing when an exception occurs:

```
try {
   solution = root(A,B,C);
}
catch (IllegalArgumentException e) {
   System.out.println("Sorry, I can't find a solution.");
   System.out.println(e.getMessage());
   continue;  // jump back to start of the while loop.
}

/* We only get to this point if NO exception was thrown.
   So we know that the solution was computed successfully.
*/
```

```
System.out.println("A solution of the equation is " +
solution);
```

The rest of the main program is pretty standard. A complete solution is shown below.

The Solution

```
import textio.TextIO;

/**
 * This program will compute and print one of the solutions
 * to an equation of the form A*X*X + B*X + C = 0, where
 * A, B, and C are numbers entered by the user.  It depends
 * on the non-standard class TextIO for doing input.
 */
public class Quadratic {

   public static void main(String[] args) {

      double A, B, C;   // Coefficients in the equation.
      double solution;  // The solution computed for the
equation.
      boolean goAgain;  // Set to true if the user wants to
                        //   solve another equation.

      System.out.println("This program will print a
solution of an equation");
      System.out.println("of the form A*X*X + B*X + C = 0,
where A, B, and");
      System.out.println("C are values that you enter.");

      do {

         /* Get the coefficients from the user. */

         System.out.println();
         System.out.println("Enter values for A, B, and
C:");
         System.out.print("A = ");
         A = TextIO.getlnDouble();
         System.out.print("B = ");
         B = TextIO.getlnDouble();
         System.out.print("C = ");
         C = TextIO.getlnDouble();
```

```
         System.out.println();

         /* Print the solution, or an error message, if
            there is no solution. */

         try {
            solution = root(A,B,C);
            System.out.println("A solution of the equation
is " + solution);
         }
         catch (IllegalArgumentException e) {
            System.out.println("Sorry, I can't find a
solution.");
            System.out.println(e.getMessage());
         }

         /* Find out whether the user wants to go again. */

         System.out.println();
         System.out.print("Do you want to solve another
equation? ");
         goAgain = TextIO.getlnBoolean();

      } while (goAgain);

   } // end main


   /**
    * 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);
       }
   }

}  // end class Quadratic
```

Solution for Programming Exercise 8.2

Exercise 8.2:

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 or with the `nextLine()` function of a *Scanner*. 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! The program should not end when that happens; it should just output an error message. 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.

Discussion

My solution uses a subroutine, `printThreeNSequence(N)`, to print out the `3N+1` sequence starting from the *BigInteger*, `N`. The subroutine assumes that `N` is not `null` and that it represents a value that is greater than one. Given these assumptions, the subroutine cannot generate any errors. (These preconditions are ensured by the main program which calls the subroutine, but I use `assert` statements to test the assumptions during debugging.) The only interesting aspect of the subroutine is that all operations on `N` must be performed using instance methods from the *BigInteger* class. For example, to multiply `N` by 2,

I use a statement "`N = N.multiply(TWO);`", where `TWO` is a *BigInteger* that represents the integer 2. My program defines `TWO` as a constant, along with several other *BigIntegers* that represent values that I need:

```
static final BigInteger THREE = new BigInteger("3");
static final BigInteger ONE = new BigInteger("1");
static final BigInteger TWO = new BigInteger("2");
```

With these constants, the code for computing the next term in a `3N+1` sequence becomes:

```
if (N.testBit(0) == false) {
        // N is even.  Divide N by 2.
    N = N.divide(TWO);
}
else {
        // N is odd.  Multiply N by 3, then add 1.
    N = N.multiply(THREE);
    N = N.add(ONE);
}
```

You can find the complete subroutine in the solution that is given below. (Note that the two lines in the else part could be replaced by one line: `N = N.multiply(THREE).add(ONE);`)

In the `main()` routine, the user's input is read into a variable, `line`, of type *String*. The input is used to construct a *BigInteger* with the statement "`N = new BigInteger(line);`". Since this statement can produce a *NumberFormatException*, it is placed in a `try` statement that can catch and handle the error. The test "`if (N.signum() == 1)`" is used to make sure that `N >= 1`. The value of `N.signum()` is 1 if and only if `N` is a positive integer. Here is the loop form the `main()` routine that processes the user's input (where `scanner` is a *Scanner* that reads from `System.in`):

```
while (true) {
   System.out.println();
   System.out.println("Enter the starting value, or press
return to end.");
   System.out.print("? ");
   line = scanner.nextLine().trim();
   if (line.length() == 0)
       break;
   try {
```

```
        N = new BigInteger(line);
        if (N.signum() == 1)
           printThreeNSequence(N);
        else
           System.out.println("Error:  The starting value
cannot be less than or equal to zero.");
    }
    catch (NumberFormatException e) {
        System.out.println("Error:  \"" + line + "\" is not
a legal integer.");
    }
}
```

Note that the user's input is read using "`line = scanner.nextLine().trim()`. The function `trim()` that is applied to the input string will remove any spaces that are at the start or the end of the string. In the constructor `new BigInteger(line)`, no spaces are allowed in `line`, but if the user types a space or two followed by a legal number, I don't want my program to reject that as an error. Using the `trim()` command will allow spaces before and after an otherwise legal number. Using `scanner.next()` instead of `scanner.nextLine()` would be another way to ignore spaces, but it would make it impossible to end the program when the user enters an empty line because `scanner.next()` just skips empty lines while looking for the next non-blank token.

The Solution

```
import java.math.BigInteger;
import java.util.Scanner;

/**
 * This program prints out 3N+1 sequences for starting
values of N that
 * are entered by the user.  Since integers are represented
as objects of
 * type BigInteger, it will work for arbitrarily large
integers.  The
 * starting value specified by the user must be greater
than zero.  The
 * program continues to read input from the user and print
3N+1 sequences
 * until the user inputs an empty line.  If the user's
```

```
input is illegal,
 * the program will print an error message and continue.
 */
public class BigThreeN {


    private static final BigInteger THREE = new
BigInteger("3");
    private static final BigInteger ONE = new
BigInteger("1");
    private static final BigInteger TWO = new
BigInteger("2");


    public static void main(String[] args) {

       Scanner scanner = new Scanner( System.in );  // for
reading user's input.

       String line;   // A line of input from the user.

       BigInteger N;  // The starting point for the 3N+1
sequence,
                      //   as specified by the user.

       System.out.println("This program will print 3N+1
sequences for positive starting values");
       System.out.println("that you enter.  There is no
pre-set limit on the number of");
       System.out.println("digits in the numbers that you
enter.  The program will end when");
       System.out.println("you enter an empty line.");

       while (true) {
          System.out.println();
          System.out.println("Enter the starting value, or
press return to end.");
          System.out.print("? ");
          line = scanner.nextLine().trim();
          if (line.length() == 0)
              break;
          try {
              N = new BigInteger(line);
              if (N.signum() == 1)
                 printThreeNSequence(N);
              else
                 System.out.println("Error:  The starting
value cannot be less than or equal to zero.");
```

```
          }
          catch (NumberFormatException e) {
              System.out.println("Error:  \"" + line + "\"
is not a legal integer.");
          }
       }

       System.out.println();
       System.out.println("OK.  Bye for now!");

    }  // end main()


    /**
     * Print the 3N+1 sequence starting from N, and count
the number
     * of terms in the sequence.  It is assumed that N is
not null and
     * that it is greater than zero.
     */
    private static void printThreeNSequence(BigInteger N) {

       assert N != null && N.signum() == 1 : "Illegal
parameter value.";

       int count;  // The number of terms in the sequence.

       System.out.println();
       System.out.println("The 3N+1 sequence starting with
" + N + " is:");
       System.out.println();

       System.out.println(N.toString());   // Print N as
the first term of the sequence
       count = 1;

       while ( ! N.equals(ONE) ){   // Compute and print
the next term
          if (N.testBit(0) == false) {
                  // N is even.  Divide N by 2.
              N = N.divide(TWO);
          }
          else {
                  // N is odd.  Multiply N by 3, then add
1.
              N = N.multiply(THREE);
              N = N.add(ONE);
          }
```

```
            System.out.println(N.toString());
            count++;
        }

        System.out.println();
        System.out.println("There were " + count + " terms
in the sequence.");

    }  // end printThreeNSequence


} // end BigThreeN
```

Solution for Programming Exercise 8.3

Exercise 8.3:

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.

Discussion

My class is called *RomanNumeral*. An object of type *RomanNumeral* has a `private` instance variable of type int that stores the integer value of the Roman numeral. When the `toString()` method is called, it computes the string that represents the Roman number based on the value of this int. By contrast, the `toInt()` method simply returns the value of the instance variable. This is not the only way that things could be done. I might have stored the string representation of the Roman numeral in an instance variable. In that case, the `toString()` method would simply return the stored value, while the `toInt()` method would have to compute the int value from the stored *String*. It would also be possible, and more efficient, to store both the int and *String* representations in the object. (The point, though, is that it's not necessary to do so. The two representations hold exactly the same information.)

In my version of the class, the constructor which takes a parameter of type int simply has to store the parameter value in the instance variable, after checking that it is in the legal range of values.

The constructor that takes a parameter of type *String* must interpret the string as a Roman numeral and convert it to the corresponding int value. This is done by adding up the integer value associated with each character or pair of characters in the string. The fact that characters sometimes need to be considered in pairs complicates things a bit. An algorithm for converting a *String*, `roman`, to an int, `arabic`, is:

```
Let arabic = 0
Let i = 0  // representing a position in the string

while i is a legal position in the string:
    Let ch be the character in position i
    Let N be the numeric equivalent of ch
    i++   // to account for the character, ch
    if there are no additional characters in the string:
          // (We need to make this test first, to avoid an
error
          // when we try to look at the next character.)
       Add N to arabic
```

```
    else:   // Try pairing the ch with the next character
       Let N2 be the numeric equivalent of the NEXT
character
       If N < N2:  // Evaluate the characters as a pair
           Add (N2 - N) to arabic
           i++    // to account for the extra character
       else:
           Add N to arabic
```

This algorithm does not take into account that the string might not be a legal Roman numeral. If a character in the string is not one of the characters M, D, C, L, X, V, or I, then a *NumberFormatException* must be thrown. Note that the algorithm allows some unusual two-letter combinations such as IM to represent 999, and it takes IIIIII to represent 6. It even allows IIM to represent 1000, which is even stranger. It is not clear whether these things should be considered to be errors. My program accepts strings like these as input. If the input is legal according to the algorithm but is not in the usual form, the program outputs the roman numeral in standard form.

The job of converting an int into an equivalent Roman numeral is handled in my `toString()` method. The exercise includes code that shows how to write this method as a long sequence of `while` loops. Consider the loop

```
while (N >= 1000) {
   roman += "M";
   N -= 1000;
}
```

After this loop, all the 1000's in `N` have been converted to M's in `roman`, and we can be sure that `N` is 999 or less. So what's left of `N` can be expressed in terms of the smaller numbers in the table: 900, 500, 400, and so on. Each of these numbers can be processed by a `while` loop (although an if statement would also work in some cases, where the number that is being tested can only occur once.). Note that the numbers in these loops must be in decreasing order for this to work.

However, all the loops in this algorithm have the same form. They just use different numbers and letters. In my program, I use two arrays to store the numbers and letters from the table:

```
private static int[]    numbers = { 1000,  900,  500,
400,  100,   90,
                                        50,   40,   10,
9,    5,    4,    1 };

private static String[] letters = { "M",  "CM",  "D",
"CD", "C",  "XC",
                                      "L",  "XL",  "X",
"IX", "V",  "IV", "I" };
```

For each index `i`, `numbers[i]` is the int equivalent of the Roman numeral `letters[i]`. All the processing can then be done with a `for` loop that does all the required `while` loops one after the other:

```
public String toString() {
      // Return the standard representation of this Roman
numeral.
   String roman = "";  // The Roman numeral.
   int N = num;        // N represents the part of num
that still has
                       //   to be converted to Roman
numeral representation.
   for (int i = 0; i < numbers.length; i++) {
      while (N >= numbers[i]) {
         roman += letters[i];
         N -= numbers[i];
      }
   }
   return roman;
}
```

An algorithm for the main program is given by:

```
while (true):
    Prompt the user for input
    If the first non-blank thing on the line is the end-
of-line:
       break
    else if the first character on the line is a digit:
       Let arabic = TextIO.getlnInt()
       Let roman = new RomanNumeral(arabic)
       Print out roman.toString()
    else:
       Let str = TextIO.getln();
```

```
        Let roman = new RomanNumeral(str);
        Print out roman.toInt();
```

This algorithm ignores the possibility that the user's input might be illegal. If it is, then the *RomanNumeral* constructor will throw a *NumberFormatException*. This exception must be caught and handled. With this in mind, the algorithm becomes:

```
while (true):
    Prompt the user for input
    If the first non-blank thing on the line is the end-
of-line:
       break
    else if the first character on the line is a digit:
       Let arabic = TextIO.getlnInt()
       try {
          Let roman = new RomanNumeral(arabic)
          Print out roman.toString()
       }
       catch (NumberFormatException e) {
          Print an error message
       }
    else:
       Let str = TextIO.getln();
       try {
          Let roman = new RomanNumeral(str);
          Print out roman.toInt();
       }
       catch (NumberFormatException e) {
          Print an error message
       }
```

This can be easily coded into Java. By the way, the test as to whether the first character on the input line is a digit can be performed using the standard boolean-valued function `Character.isDigit(ch)`, which returns `true` if the character `ch` is a digit.

One final note: The *RomanNumeral* class uses a `switch` statement to convert individual characters to their numerical equivalent. The `switch` statement uses the older syntax:

```
private int letterToNumber(char letter) {
   switch (letter) {
      case 'I':  return 1;
```

```
      case 'V':  return 5;
      case 'X':  return 10;
      case 'L':  return 50;
      case 'C':  return 100;
      case 'D':  return 500;
      case 'M':  return 1000;
      default:   throw new NumberFormatException(
                   "Illegal character \"" + letter + "\"
in Roman numeral");
   }
}
```

This method could also be written using a switch expression (see Subsection 3.6.5):

```
private int letterToNumber(char letter) {
   return switch (letter) {
      case 'I' ->  1;
      case 'V' ->  5;
      case 'X' ->  10;
      case 'L' ->  50;
      case 'C' ->  100;
      case 'D' ->  500;
      case 'M' ->  1000;
      default  ->  throw new NumberFormatException(
                   "Illegal character \"" + letter + "\"
in Roman numeral");
   };
}
```

The Solution

The Roman numerals class:

```
    /**
     * An object of type RomanNumeral is an integer between
1 and 3999.  It can
     * be constructed either from an integer or from a
string that represents
     * a Roman numeral in this range.  The function
toString() will return a
     * standardized Roman numeral representation of the
number.  The function
     * toInt() will return the number as a value of type
```

```
int.
     */
    public class RomanNumeral {

       private final int num;   // The number represented
by this Roman numeral.

       /* The following arrays are used by the toString()
function to construct
        * the standard Roman numeral representation of the
number.  For each i,
        * the number numbers[i] is represented by the
corresponding string, letters[i].
        */

       private static int[]    numbers = { 1000,  900,
500,  400,  100,   90,
                                                50,   40,
10,    9,    5,    4,    1 };

       private static String[] letters = { "M",  "CM",
"D",  "CD", "C",  "XC",
                                              "L",  "XL",
"X",  "IX", "V",  "IV", "I" };

       /**
        * Constructor.  Creates the Roman number with the
int value specified
        * by the parameter.
        * @throws NumberFormatException if the parameter is
not in the range 1 to 3999 inclusive.
        */
       public RomanNumeral(int arabic) {
          if (arabic < 1)
             throw new NumberFormatException("Value of
RomanNumeral must be positive.");
          if (arabic > 3999)
             throw new NumberFormatException("Value of
RomanNumeral must be 3999 or less.");
          num = arabic;
       }


       /*
        * Constructor.  Creates the Roman number with the
given representation.
        * For example, RomanNumeral("xvii") is 17.  If the
parameter is not a
```

```
        * legal Roman numeral, a NumberFormatException is
thrown.  Both upper and
        * lower case letters are allowed.
        * @throws NumberFormatException if the parameter is
not a legal Roman numeral
        */
       public RomanNumeral(String roman) {

          if (roman.length() == 0)
             throw new NumberFormatException("An empty
string does not define a Roman numeral.");

          roman = roman.toUpperCase();  // Convert to upper
case letters.

          int i = 0;       // A position in the string,
roman;
          int arabic = 0;  // Arabic numeral equivalent of
the part of the string that has
                           //    been converted so far.

          while (i < roman.length()) {

             char letter = roman.charAt(i);        //
Letter at current position in string.
             int number = letterToNumber(letter);  //
Numerical equivalent of letter.

             i++;  // Move on to next position in the
string

             if (i == roman.length()) {
                   // There is no letter in the string
following the one we have just processed.
                   // So just add the number corresponding
to the single letter to arabic.
                arabic += number;
             }
             else {
                   // Look at the next letter in the
string.  If it has a larger Roman numeral
                   // equivalent than number, then the two
letters are counted together as
                   // a Roman numeral with value
(nextNumber - number).
                int nextNumber =
letterToNumber(roman.charAt(i));
                if (nextNumber > number) {
```

```
                    // Combine the two letters to get one
value, and move on to next position in string.
                   arabic += (nextNumber - number);
                   i++;
                }
                else {
                    // Don't combine the letters.  Just
add the value of the one letter onto the number.
                   arabic += number;
                }
             }

          }  // end while

          if (arabic > 3999)
             throw new NumberFormatException("Roman numeral
must have value 3999 or less.");

          num = arabic;

       } // end constructor


       /**
        * Find the integer value of letter considered as a
Roman numeral.  Throws
        * NumberFormatException if letter is not a legal
Roman numeral.  The letter
        * must be upper case.
        */
       private int letterToNumber(char letter) {
          switch (letter) {
             case 'I':  return 1;
             case 'V':  return 5;
             case 'X':  return 10;
             case 'L':  return 50;
             case 'C':  return 100;
             case 'D':  return 500;
             case 'M':  return 1000;
             default:   throw new NumberFormatException(
                          "Illegal character \"" + letter +
"\" in Roman numeral");
          }
       }


       /**
        * Return the standard representation of this Roman
```

```
numeral.
        */
       public String toString() {
          String roman = "";  // The roman numeral.
          int N = num;        // N represents the part of
num that still has
                              //   to be converted to Roman
numeral representation.
          for (int i = 0; i < numbers.length; i++) {
             while (N >= numbers[i]) {
                roman += letters[i];
                N -= numbers[i];
             }
          }
          return roman;
       }


       /**
        * Return the value of this Roman numeral as an int.
        */
       public int toInt() {
          return num;
       }


    } // end class RomanNumeral
```

The main program class:

```
import textio.TextIO;

    /**
     * This program will convert Roman numerals to ordinary
arabic numerals
     * and vice versa.  The user can enter a numerals of
either type.  Arabic
     * numerals must be in the range from 1 to 3999
inclusive.  The user ends
     * the program by entering an empty line.
     */
    public class RomanConverter {

       public static void main(String[] args) {

          System.out.println("Enter a Roman numeral and I
```

```
will convert it to an ordinary");
          System.out.println("arabic integer.  Enter an
integer in the range 1 to 3999");
          System.out.println("and I will convert it to a
Roman numeral.  Press return when");
          System.out.println("you want to quit.");

          while (true) {

             System.out.println();
             System.out.print("? ");

             /* Skip past any blanks at the beginning of
the input line.
                Break out of the loop if there is nothing
else on the line. */

             while (TextIO.peek() == ' ' || TextIO.peek()
== '\t')
                TextIO.getAnyChar();
             if ( TextIO.peek() == '\n' )
                break;

             /* If the first non-blank character is a
digit, read an arabic
                numeral and convert it to a Roman numeral.
Otherwise, read
                a Roman numeral and convert it to an arabic
numeral. */

             if ( Character.isDigit(TextIO.peek()) ) {
                int arabic = TextIO.getlnInt();
                try {
                    RomanNumeral N = new
RomanNumeral(arabic);
                    System.out.println(N.toInt() + " = " +
N.toString());
                }
                catch (NumberFormatException e) {
                    System.out.println("Invalid input.");
                    System.out.println(e.getMessage());
                }
             }
             else {
                String roman = TextIO.getln();
                try {
                    RomanNumeral N = new
RomanNumeral(roman);
```

```
                    System.out.println(N.toString() + " = "
+ N.toInt());
                    if ( !
roman.equalsIgnoreCase(N.toString()) ) {
                        System.out.printf("In standard
form, %s is written %s.%n",
                                                  roman,
N.toString());
                    }
                }
                catch (NumberFormatException e) {
                    System.out.println("Invalid input.");
                    System.out.println(e.getMessage());
                }
            }

        }  // end while

        System.out.println("OK.  Bye for now.");

    }  // end main()

} // end class RomanConverter
```

Solution for Programming Exercise 8.4

Exercise 8.4:

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.

Discussion

A pseudocode algorithm for the program is given by:

```
while (true):
    Get a line of input from the user
    if the line is empty:
       break
    Convert the input line to an Expr
    Read and process the user's numbers
```

Converting the input line into an object of type *Expr* involves calling the constructor from the *Expr* class. This call might generate an *IllegalArgumentException*. The algorithm must be expanded to handle this exception and print an error message if it occurs. When an error occurs, I use a `continue` statement to jump back to the start of the loop without reading any numbers from the user:

```
while (true):
    Get a line of input from the user
    if the line is empty:
       break
    try {
       Let expression = new Expr(line)
    }
    catch (IllegalArgumentException e) {
       Print an error message
       continue   // jumps back to start of loop
    }
    Read and process the user's numbers
```

The last step, reading and processing the user's numbers, expands into a loop, which is nested inside the main `while` loop. In this loop, I could use `TextIO.getDouble()` to read one of the user's numbers, but instead I chose

to read the user's input into a string and convert that string into a value of type double. This has two advantages: I can end the loop when the user presses return. And I can do nicer error handling than the default error-handling that is provided by *TextIO*. The conversion from a string, `line`, to a double is done using a method `Double.parseDouble(string)`. The conversion will generate a *NumberFormatException* if the user's input is not a legal number. The algorithm for reading and processing the user's numbers becomes:

```
while (true):
    Get a line of input from the user
    if the line is empty:
       break
    try {
       Let x = Double.parseDouble(line)
    }
    catch (NumberFormatException e) {
       Print an error message
       continue
    }
    Let val = expression.value(x)
    if val is Double.NaN:
       Print an error message
    else:
       Output val
```

All this can be easily translated into the complete solution, which follows.

The Solution

```
import textio.TextIO;

public class FunctionEvaluator {

   public static void main(String[] args) {

      String line;       // A line of input read from the
user.
      Expr expression;   // The definition of the function
f(x).
      double x;          // A value of x for which f(x) is
to be calculated.
      double val;        // The value of f(x) for the
specified value of x.
```

```
      System.out.println("This program will evaluate a
specified function, f(x), at");
      System.out.println("specified values of the variable
x.  The definition of f(x)");
      System.out.println("can use the operators +, -, *, /,
and ^ as well as mathematical");
      System.out.println("functions such as sin, abs, and
ln.");

      while (true) {

         /* Get the function from the user.  A line of
input is read and
            used to construct an object of type Expr.  If
the input line is
            empty, then the loop will end, and the program
will terminate. */

         System.out.println("\n\n\nEnter definition of
f(x), or press return to quit.");
         System.out.print("\nf(x) = ");
         line = TextIO.getln().trim();
         if (line.length() == 0)
            break;

         try {
            expression = new Expr(line);
         }
         catch (IllegalArgumentException e) {
                // An error was found in the input.  Print
an error
                //    message and go back to the beginning
of the loop.
            System.out.println("Error!  The definition of
f(x) is not valid.");
            System.out.println(e.getMessage());
            continue;
         }

         /* Read values of x from the user, until the user
presses return.
            If the user's input is not a legal number,
print an error message.
            Otherwise, compute f(x) and print the result.
*/

         System.out.println("\nEnter values of x where f(x)
```

```
is to be evaluated.");
         System.out.println("Press return to end.");

         while (true) {
            System.out.print("\nx = ");
            line = TextIO.getln().trim();
            if (line.length() == 0)
               break;
            try {
               x = Double.parseDouble(line);
            }
            catch (NumberFormatException e) {
               System.out.println("\"" + line + "\" is not
a legal number.");
               continue;
            }
            val = expression.value(x);
            if (Double.isNaN(val))
               System.out.println("f(" + x + ") is
undefined.");
            else
               System.out.println("f(" + x + ") = " + val);
         }  // end while

      } // end while

      System.out.println("\n\n\nOK.  Bye for now.");

   }  // end main();

} // end class FunctionEvaluator
```

Solution for Programming Exercise 8.5

Exercise 8.5:

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 clicks an "Enter" button or presses return, 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. (Recall: If you make a button into the default button for the window, then pressing return will be equivalent to clicking the button (see the end of Subsection 6.4.2).)

The program will need a *Canvas* for displaying the graph. To keep things simple, the canvas 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 x and y values in the range from -5 to 5 to the pixel coordinates that you need for drawing on the canvas. The formulas for the conversion are:

```
double a  =  ( (x + 5)/10 * width );
double b  =  ( (5 - y)/10 * height );
```

where `a` and `b` are the horizontal and vertical pixel coordinates on the canvas. The values of `width` and `height` give the size of the canvas.

Discussion

I wrote my solution using a nested subclass, *GraphCanvas*, of *Canvas* to represent the canvas where the function is graphed. The subclass has an instance variable, `func`, of type *Expr* that represents the function to be drawn, with a setter method that can be called to change the function. The value of `func` can be set to `null` to indicate that no function is available to be graphed. This will be true in the program when the user's input has been found to be illegal. The subclass has a `draw()` method that checks the value of `func` to decide what to draw. If `func` is `null`, then the `draw()` method simply draws a message on the panel stating that no function is available. Otherwise, it draws a pair of axes and the graph of the function, and it displays the definition of the function as a string.

The interesting work in class *GraphCanvas* is done in the `drawFunction()` method, which is called by `draw()`. This function draws the graph of the function for `-5 <= x <= 5`. This interval on the `x` axis is divided into 300 subintervals. Since the length of the interval is 10, the length of each subinterval is given by `dx`, where `dx` is `10.0/300`. The `x` values for the points that I want to plot are given by `-5`, `-5+dx`, `-5+2*dx`, and so on. Each `x`-value is obtained by adding `dx` to the previous value. For each `x` value, the `y`-value of the point on the graph is computed as `func.value(x)`. As the points on the graph are computed, line segments are drawn to connect pairs of points (unless the `y`-value of either point is undefined). An algorithm for the `drawFunction()` method is:

```
Let dx = 10.0 / 300;
Let x = -5                  // Get the first point
Let y = func.value(x)
for i = 1 to 300:
    Let prevx = x           // Save the previous point
    Let prevy = y
    Let x = x + dx          // Get the next point
    Let y = func.value(x)
```

```
    if neither y nor prevy is Double.NaN:
       draw a line segment from (prevx,prevy) to (x,y)
```

The method for drawing the line segment uses the conversion from graphing coordinates to pixel coordinates that is given in the exercise. By the way, more general conversion formulas can be given in the case where `x` extends from `xmin` to `xmax` and `y` extends from `ymin` to `ymax`. The general formulas are:

```
a  =  ( (x - xmin) / (xmax - xmin) * width );
b  =  ( (ymax - y) / (ymax - ymin) * height );
```

The formulas for `a` and `b` are of slightly different form to reflect the fact that `a` increases from 0 to `width` as `x` increases from `xmin` to `xmax`, while `b` **decreases** from `height` to 0 as `y` increases from `ymin` to `ymax`. You could improve the program by adding text input boxes where the user can enter values for `xmin`, `xmax`, `ymin`, and `ymax`.

The constructor for the *GraphCanvas* takes a parameter of type *Expr* that specifies an initial function to be graphed when the canvas is first created (or the parameter could be `null` if no initial function is to be displayed). I use this feature in my program to show a sample function when the program starts.

In the `start()` method, a *BorderPane* is used for the root of the scene graph. The center component is the large canvas that is used for graphing. A *Label* is placed in the top position; the label is used to display messages to the user, including error messages when the user's input is not valid. The bottom component is an *HBox* that contains the text input box for the user's input and the "Enter" button. I also added a label in front of the text input box that says "f(x) =". The start method also installs an `ActionEvent` handler for the button, written as a long lambda expression. I did a great deal of tweaking of the user interface, using techniques from Chapter 6, but since this chapter is not about GUI programming, I won't discuss the tweaks here. You can check out the source code below to see what I did. Here is what the program window looks like when it first appears on the screen:

The action event handler gets the string from the textfield. It tries to use this string to construct an object of type `Expr`. That constructor throws an `IllegalArgumentException` if the string contains an error, so the constructor is called in a `try` statement that can catch and handle the error. If an error occurs, then the error message in the exception object is displayed in the label at the top of the window, and the graph is cleared. If no error occurs, the graph is set to display the user's function, and the label is set to display the generic message, "Enter a function and click Enter or press return." The code for all this is:

```
Expr function;  // The user's function.

try {
    String def = functionInput.getText();
    function = new Expr(def);
    graph.setFunction(function);
    message.setText(" Enter a function and press
return.");
}
```

```
catch (IllegalArgumentException e) {
    graph.clearFunction();
    message.setText(e.getMessage());
}
```

The Solution

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.layout.HBox;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.Priority;
import javafx.scene.control.Label;
import javafx.scene.control.Button;
import javafx.scene.control.TextField;
import javafx.scene.paint.Color;


/**
   The SimpleGrapher program can draw graphs of functions
input by the
   user.  The user enters the definition of the function in
a text
   input box.  When the user presses return or clicks the
"Enter" button,
   the function is graphed.  (Unless the definition
contains an error.
   In that case, an error message is displayed.)

   The graph is drawn on a canvas which represents the
region of the
   (x,y)-plane given by  -5 <= x <= 5  and  -5 <= y <= 5.
Any part of
   the graph that lies outside this region is not shown.
The graph
   is drawn by plotting 301 points and joining them with
lines.  This
   does not handle discontinuous functions properly.

   An example function is graphed when the program starts.

   This program requires the class Expr, which is defined
in by a
```

```
   separate file, Expr.java.  That file contains a full
description
   of the syntax of legal function definitions, but the
program
   understands operators +, -, *, /, and ^ (where ^
represents
   exponentiation), as well as common mathematical
functions
   such as sin(x) and ln(x).
 */

public class SimpleGrapher extends Application {

    public static void main(String[] args) {
        launch(args);
    }

    //-----------------------------------------------------
----------------------------

    private GraphCanvas graph;  // The Canvas that will
display the graph.
                                // GraphCanvas is a
subclass of Canvas that
                                // is defined as a static
nested class.

    private TextField functionInput;  // A text input box
where the user enters
                                      // the definition of
the function.

    private Label message;  // A label for displaying
messages to the user,
                            // including error messages
when the function
                            // definition is illegal.

    /**
     * Set up the GUI with a large canvas in the center
where the functions
     * are graphed, a label at the top for displaying
messages, and
     * an input box for the function below the canvas.
Also adds an Enter
     * button with an ActionEvent handler that graphs the
function.  The
     * button is set to be the default button in the
```

```
window, so that the
     * user can also graph the function by pressing return.
     */
    public void start(Stage stage) {

        /* Create the components and set up event handling.
The
         * canvas is given an initial function to draw, and
the
         * textfield is initialized to show the definition
of
         * that function. */

        graph = new GraphCanvas( new Expr("sin(x)*3 +
cos(5*x)") );

        message = new Label(" Enter a function and click
Enter or press return");

        functionInput = new TextField("sin(x)*3 +
cos(5*x)");

        Button graphIt = new Button("Enter");
        graphIt.setDefaultButton(true);

        graphIt.setOnAction( evt -> {
                    // Get the user's function definition
from the box and use it
                    // to create a new object of type Expr.
Tell the GraphCanvas to
                    // graph this function.  If the
definition is illegal, an
                    // IllegalArgumentException is  thrown
by the Expr constructor.
                    // If this happens, the graph is
cleared and an error message
                    // is displayed in the message label.
                Expr function;  // The user's function.
                try {
                    String def = functionInput.getText();
                    function = new Expr(def);
                    graph.setFunction(function);
                    message.setText(" Enter a function and
click Enter or press return.");
                }
                catch (IllegalArgumentException e) {
                    graph.clearFunction();
                    message.setText(e.getMessage());
```

```
                }
                functionInput.selectAll();
                functionInput.requestFocus();  // Let's
user start typing in input box.
            } );

        /* Create the layout. */

        HBox bottom = new HBox(8, new Label("f(x) ="),
functionInput, graphIt);

        BorderPane root = new BorderPane();
        root.setCenter(graph);
        root.setTop(message);
        root.setBottom(bottom);

        /* Tweak the components to make the program more
attractive.  Add borders
         * around the entire root pane and between the
canvas and the top and bottom
         * components.  Adding some padding around the
components in the bottom HBox. */

        root.setStyle("-fx-border-color:gray; -fx-border-
width:4px");
        message.setTextFill(Color.RED);  // User red text
for the message.
        message.setStyle("-fx-background-color:white; -fx-
padding:7px; "
                + "-fx-border-color:gray; -fx-border-
width:0 0 4px 0");
        message.setMaxWidth(10000);  // Required to make
the label (and its border)
                                     // extend the full
width of the window.
        bottom.setStyle("-fx-border-color:gray; -fx-border-
width:4px 0 0 0; -fx-padding:8px");
        HBox.setHgrow(functionInput, Priority.ALWAYS); //
Allows functionInput to grow
                                                        //
to fill the available space.

        /* Finish setting up the window and make it
visible. */

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setResizable(false);
```

```
        stage.setTitle("A Simple Function Grapher");
        stage.show();

    }  // end start()


    //   -------------------------- Nested class ----------
------------------

    private static class GraphCanvas extends Canvas {

        // A object of this class can display the graph of
a function
        // on the region of the (x,y)-plane given by -5 <=
x <= 5 and
        // -5 <= y <= 5.  The graph is drawn very simply,
by plotting
        // 301 points and connecting them with line
segments.  The canvas
        // is 600-by-600 pixels.  The size could be changed
by editing
        // the definition in the constructor.

        Expr func;  // The definition of the function that
is to be graphed.
                    // If the value is null, no graph is
drawn.


        GraphCanvas(Expr firstFunction) {
            super(600,600);  // Calls the constructor from
the Canvas class.
            func = firstFunction;
            draw();  // Draw the canvas at startup.
        }


        public void setFunction(Expr exp) {
                // Set the canvas to graph the function
whose definition is
                // given by the function exp.
            func = exp;
            draw();
        }


        public void clearFunction() {
                // Set the canvas to draw no graph at all.
```

```
            func = null;
            draw();
        }


        public void draw() {
                // Fill the canvas with white, then draw a
set of axes
                // and the graph of the function.  Or, if
func is null,
                // display a message that there is no
function to be graphed.
            GraphicsContext g = getGraphicsContext2D();
            g.setFill(Color.WHITE);
            g.fillRect(0,0,getWidth(),getHeight());

            if (func == null) {
                g.setFill( Color.RED );
                g.fillText("No function is available.", 30,
40);
            }
            else {
                g.setFill( Color.PURPLE );
                g.fillText("y = " + func, 5, 15);
                drawAxes(g);
                drawFunction(g);
            }
        }


        void drawAxes(GraphicsContext g) {
                // Draw horizontal and vertical axes in the
middle of the
                // canvas.  A 5-pixel border is left at the
ends of the axes.
            double width = getWidth();
            double height = getHeight();
            g.setStroke(Color.BLUE);
            g.setLineWidth(2);
            g.strokeLine(5, height/2, width-5, height/2);
            g.strokeLine(width/2, 5, width/2, height-5);
        }


        void drawFunction(GraphicsContext g) {
                // Draw the graph of the function defined
by the instance
                // variable func.  Just plot 301 points
```

```
with lines
                // between them. s

            double x, y;          // A point on the graph.
y is f(x).
            double prevx, prevy;  // The previous point on
the graph.

            double dx;  // Difference between the x-values
of consecutive
                        // points on the graph.

            dx  = 10.0 / 300;

            g.setStroke(Color.RED);
            g.setLineWidth(1);

            /* Compute the first point. */

            x = -5;
            y = func.value(x);

            /* Compute each of the other 300 points, and
draw a line segment
               between each consecutive pair of points.
Note that if the
               function is undefined at one of the points
in a pair, then
               the line segment is not drawn.  */

            for (int i = 1; i <= 300; i++) {

                prevx = x;           // Save the coords of
the previous point.
                prevy = y;

                x += dx;            // Get the coords of
the next point.

                y = func.value(x);

                if ( (! Double.isNaN(y)) && (!
Double.isNaN(prevy)) ) {
                       // Draw a line segment between the
two points.
                    putLine(g, prevx, prevy, x, y);
                }
```

```
            }  // end for

        }  // end drawFunction()


        void putLine(GraphicsContext g, double x1, double
y1,
                double x2, double y2) {
                    // Draw a line segment from the point
(x1,y1) to (x2,y2).
                    // These values must be scaled to
convert from coordinates
                    // that go from -5 to 5 to the
coordinates that are needed
                    // for drawing on the canvas, which go
from 0 to 600.
                    // coordinates of the corresponding
pixels.

            if (Math.abs(y1) > 10000 || Math.abs(y2) >
10000) {
                    // Only draw lines for reasonable y-
values.
                    // This should not be necessary, but
I'm not sure
                    // how GraphicsContext will handle very
large values.
                return;
            }

            double a1, b1;   // Pixel coordinates
corresponding to (x1,y1).
            double a2, b2;   // Pixel coordinates
corresponding to (x2,y2).

            double width = getWidth();     // Width of the
canvas (600).
            double height = getHeight();   // Height of the
canvas (600).

            a1 = (int)( (x1 + 5) / 10 * width );
            b1 = (int)( (5 - y1) / 10 * height );
            a2 = (int)( (x2 + 5) / 10 * width );
            b2 = (int)( (5 - y2) / 10 * height );

            g.strokeLine(a1,b1,a2,b2);

        }  // end putLine()
```

```
    }  // end nested class GraphCanvas

} // end class SimpleGrapher
```

Solution for Programming Exercise 9.1

Exercise 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. Consider using the *BigInteger* class (see Exercise 8.2)

(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.)

Discussion

The recursive definitions of the two functions can be translated rather easily into Java code. Note that for `factorial(N)`, the base case is `N=0`, while `fibonacci(N)` has two base cases, `N=0` and `N=1`. The obvious recursive function for computing `fibonacci(N)` is:

```
static int fibonacci( int N ) {
    if ( N == 0 || N == 1 ) {
           // Base cases; the answer is 1.
        return 1;
```

```
    }
    else {
           // Recursive case; the answer is obtained by
applying the function
           // recursively to N-1 and to N-2, and adding
the two answers.
        return fibonacci(N-1) + fibonacci(N-2);
    }
}
```

and for `factorial(N)` is

```
static int factorial( int N ) {
   if ( N == 0) {
          // Base case; the answer is 1.
      return 1;
   }
   else {
         // Recursive case; the answer is obtained by
applying the function
         // recursively to N-1 and multiplying the answer
by N.
      return factorial(N-1) * N;
   }
}
```

Adding a `main()` routine that calls these methods for several values of `N` would be an acceptable solution to the exercise. However, when testing the program, you will quickly notice some problems. First of all, the recursive algorithm for `fibonacci(N)` is so inefficient that it takes an unreasonably long time for it to run even for values of `N` as small as 40. (In fact, the recursive evaluation of `fibonacci(N)` calls both `fibonacci(N-1)` and `fibonacci(N-2)` and so takes almost twice as long as the evaluation of `fibonacci(N-1)`; with the computation time for `fibonacci(N)` almost doubling every time `N` goes up by 1, the computation time quickly becomes unreasonable.) Although I include the recursive version of `fibonacci` in my solution, my program refuses to use this function if `N` is greater than 40.

Another problem occurs because of the limited size of values of type `int`. The value of `factorial(N)` grows very quickly as `N` increases—so quickly that by the time `N` reaches 13, the value of `factorial(N)` is already too large to be expressed as a 32-bit integer! So, the `factorial` function as defined

above only gives the correct answer for numbers 0 through 12. Although `fibonacci(N)` does not grow nearly so fast as `factorial(N)`, it is still true that by the time `N` reaches 46, `fibonacci(N)` is outside the range of 32-bit integers.

The problem of the limited size of values of type int was discussed in Subsection 8.1.3. One approach to dealing with the problem was presented in Exercise 8.2: The class `java.math.BigInteger` represents integer values that can be arbitrarily large (within the limits of the computer's memory). In my solution to the exercise, I decided to use *BigInteger* values to compute `factorial(N)` and to compute `fibonacci(N)` non-recursively. This allows my program to work even for fairly large values of `N`, say up to a few thousand.

You can read my solution below to see how I used the *BigInteger* class and how I computed `fibonacci(N)` non-recursively.

The Solution

```
import java.util.Scanner;
import java.math.BigInteger;

/**
 * Computes factorial(N) and fibonacci(N) for integers N
entered by
 * the user, as a demonstration of recursion.
 */
public class FibonacciAndFactorial {

   /**
    * Main routine reads integers N from the user and
prints the
    * values of factorial(N) and fibonacci(N), stopping
when the
    * user inputs a zero.
    */
   public static void main( String[] args ) {
      Scanner in = new Scanner( System.in );
      while (true) {
         System.out.print("\n\nEnter a positive integer, or
0 to end:  ");
```

```
         int N = in.nextInt();
         if (N == 0)
            break;
         else if (N < 0) {
            System.out.println("Negative numbers are not
allowed.");
            continue;
         }
         BigInteger NasBigInteger = BigInteger.valueOf(N);
         System.out.println("\n  factorial(" + N + ") is "
+ factorial(NasBigInteger));
         if (N > 40) {
            System.out.println("\n  N is too big to compute
fibonacci(N) recursively");
         }
         else {
            System.out.println("\n  fibonacci(" + N + ") is
" +
                  fibonacci(N) + "   (recursively)");
         }
         System.out.println("\n  fibonacci(" + N + ") is "
+
                  fibonacci_nonrecursive(N) + "   (non-
recursively)");
      }
   }


   /**
    * Compute fibonacci(N) using recursion.  Because this
is so inefficient,
    * even for fairly small values of N, N should be less
than or equal to 40.
    * Also, N must be greater than or equal to zero, or an
infinite recursion
    * will occur.
    */
   static int fibonacci( int N ) {
       assert N >= 0 : "fibonacci(n) is only defined for
non-negative n";
       assert N <= 40 : "n is too large to compute
fibonacci(N) recursively";
                // (for assertions, see Subsection 8.4.1)
       if ( N == 0 || N == 1 ) {
                // Base cases; the answer is 1.
           return 1;
       }
       else {
```

```
              // Recursive case; the answer is obtained by
applying the function
              // recursively to N-1 and to N-2, and adding
the two answers.
           return fibonacci(N-1) + fibonacci(N-2);
       }
   }


   /**
    * Compute fibonacci(N) using a for loop.  The answer is
returned as
    * a BigInteger and can be very large even for fairly
small values
    * of N.  N must be greater than or equal to zero.
    */
   static BigInteger fibonacci_nonrecursive( int N ) {
      assert N >= 0 : "fibonacci(n) is only defined for
non-negative n";
      if (N == 0 || N == 1) {
             // fibonacci(0) = fibonacci(1) = 1;
         return BigInteger.ONE;
      }
      else {
         BigInteger f0 = BigInteger.ONE;  // In the loop,
this is fibonacci(i-2)
         BigInteger f1 = BigInteger.ONE;  // In the loop,
this is fibonacci(i-1)
         for (int i = 2; i <= N; i++) {
            BigInteger fi = f0.add(f1);  // Computes
fibonacci(i)
            f0 = f1;  // Update to account for i++
            f1 = fi;  // Update to account for i++
         }
         return f1;  // Final value of f1 is fibonacci(N)
      }
   }


   /**
    * Compute factorial(N) using recursion.  The
computation is done using
    * BigIntegers and can be very large even for fairly
small values of N.
    * N must be greater than or equal to zero.
    */
   static BigInteger factorial( BigInteger N ) {
      assert N.signum() >= 0 : "factorial(n) is only
```

```
defined for non-negative n";
       if ( N.equals(BigInteger.ZERO) ) {
              // Base case; the answer is 1.
           return BigInteger.ONE;
       }
       else {
              // Recursive case; the answer is obtained by
applying the function
              // recursively to N-1 and multiplying the
answer by N.
           BigInteger factorialOfNMinus1 =
factorial(N.subtract(BigInteger.ONE));
           return N.multiply(factorialOfNMinus1);
       }
   }

}
```

Solution for Programming Exercise 9.2

Exercise 9.2:

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.

Discussion

In my solution to Exercise 7.6, words are added to an arraylist in the order in which they are encountered. After the file has been completely read, the arraylist is sorted into alphabetical order before the list of words is printed. Since a binary sort tree is designed to store words in alphabetical order at all times, there is no need for sorting. At the end of the program, an inorder traversal of the tree can be used to output the words to the file. Using an inorder traversal guarantees that the words will be output in increasing order.

For my solution to this exercise, I copied the routines `treeInsert`, `treeContains`, and `countNodes` from SortTreeDemo.java. I also copied the declaration of `root` as a static member variable, since that's the variable that represents the tree itself. (It's unfortunate that `root` has to be a global variable rather than a local variable in `main()`, but it's used as a global variable in the `treeInsert` routine. A better solution to the exercise would define a *BinarySortTree* class to encapsulate the data and routines needed to represent the tree and to use a variable of type *BinarySortTree* in the program.)

Only a few changes are needed in the `main()` routine of the original program. They are shown in red italic in the solution given below. All-in-all, the

substitution of the binary tree for the arraylist is very straightforward.

The Solution

```
import textio.TextIO;

/**
 * Makes an alphabetical list of all the words in a file
selected
 * by the user.  The list can be written to a file.
 * The words are stored in a binary sort tree.
 */
public class ListAllWordsFromFileWithTree {

   private static TreeNode root;  // Pointer to the root
node in a binary tree.
                                  // This tree is used in
this program as a
                                  // binary sort tree.
When the tree is empty,
                                  // root is null (as it is
initially).



   public static void main(String[] args) {

      System.out.println("""
                  This program will ask you to select an
input file
              It will read that file and make an
alphabetical
              list of all the words in the file.  After
reading
              the file, the program asks you to select an
output
              file.  If you select a file, the list of
words will
              be written to that file; if you cancel, the
list
              be written to standard output.  All words are
converted
              to lower case, and duplicates are eliminated
from the list.
```

```
              Press return to begin.

              """);

      TextIO.getln();  // Wait for user to press return.

      try {
         if (TextIO.readUserSelectedFile() == false) {
            System.out.println("No input file selected.
Exiting.");
             System.exit(1);
         }
         // ArrayList<String> wordList = new
ArrayList<String>(); DELETED LINE
         String word = readNextWord();
         while (word != null) {
            word = word.toLowerCase();  // convert word to
lower case
            if ( treeContains(root,word) == false ) {
                  // This is a new word, so add it to the
tree
               treeInsert(word);
            }
            word = readNextWord();
         }
         int wordsInTree = countNodes(root);
         System.out.println("Number of different words
found in file:  "
               + wordsInTree);
         System.out.println();
         if (wordsInTree == 0) {
            System.out.println("No words found in file.");
            System.out.println("Exiting without saving
data.");
            System.exit(0);
         }
         // selectionSort(wordList);  DELETED LINE
         TextIO.writeUserSelectedFile(); // If user
cancels, output automatically
                                         // goes to
standard output.
         System.out.println(wordsInTree + " words found in
file:\n");
         treeList(root);
         System.out.println("\n\nDone.\n\n");
      }
      catch (Exception e) {
```

```
         System.out.println("Sorry, an error has
occurred.");
         System.out.println("Error Message:  " +
e.getMessage());
      }
      System.exit(0);  // Might be necessary, because of
use of file dialogs.
   }


   /**
    * 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)) {
         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 that the next character, 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 bread 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.
   }


   //------------- Binary Sort Tree data structures and
methods -------------------
   //------------- (Copied from SortTreeDemo.java) --------
------------------------

   /**
    * 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 the
specified string.
             // Note that left and right pointers are
initially null.
         item = str;
```

```
      }
   }  // end nested class TreeNode


   /**
    * 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()


   /**
    * 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()


   /**
```

```
    * Print the items in the tree in inorder, one item to a
line.
    * Since the tree is a sort tree, the output  will be in
increasing order.
    */
   private static void treeList(TreeNode node) {
      if ( node != null ) {
         treeList(node.left);             // Print items in
left subtree.
         System.out.println("  " + node.item);  // Print
item in the node.
         treeList(node.right);            // Print items in
the right subtree.
      }
   } // end treeList()


   /**
    * Count the nodes in the binary tree.
    * @param node A pointer to the root of the tree.  A
null value indicates
    * an empty tree
    * @return the number of nodes in the tree to which node
points.  For an
    * empty tree, the value is zero.
    */
   private static int countNodes(TreeNode node) {
      if ( node == null ) {
            // Tree is empty, so it contains no nodes.
         return 0;
      }
      else {
            // Add up the root node and the nodes in its
two subtrees.
         int leftCount = countNodes( node.left );
         int rightCount = countNodes( node.right );
         return  1 + leftCount + rightCount;
      }
   } // end countNodes()

}
```

Solution for Programming Exercise 9.3

Exercise 9.3:

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.

Discussion

To make any linked list from scratch, you have to create nodes one-by-one and link them together. In this case, we want to make nodes that contain copies of the items from the original list. We can run through the original list, look at each item, create a new node that contains a copy of that item, and link that new node into the reversed list that we are constructing. We just have to make sure that the nodes in the new list are in the desired order.

In fact this is pretty easy: As we run down the original list from start to finish, we need to build the reversed list from back to front. The first item in the original list should be at the back of the reversed list, the second item from the original goes in front of that item, and so on. This is equivalent to "pushing" the items onto the reversed list, using the same push operation that is used for a stack. An algorithm for this is:

```
Let rev be an empty list
for each item in the original list:
    Push the item onto rev
    Move on to the next item
```

This can be coded into the subroutine we need as follows:

```
/**
 * Return a new list containing the same items as the
list,
 * but in the reverse order.
 */
static ListNode reverse( ListNode list ) {
   ListNode rev = null;     // rev will be the reversed
list.
   ListNode runner = list;  // For running through the
nodes of list.
   while (runner != null) {
          // "Push" the next node of list onto the front
of rev.
      ListNode newNode = new ListNode();
      newNode.item = runner.item;
      newNode.next = rev;
      rev = newNode;
         // Move on to the next node in the list.
      runner = runner.next;
   }
   return rev;
} // end reverse()
```

The exercise lets you design your own routine for testing the subroutine. It should be tested on several lists, including an empty list. It's important to test it on the empty list since a `null` pointer often represents a special case in an algorithm, and is therefore a common source of bugs. It's also a good idea to test a list of length one, for similar reasons. In my `main()` routine, I build up a random list one node at a time and test the `reverse()` subroutine on the list at each step. The `main()` routine was probably harder to write than the subroutine!

The Solution

```
/**
 * This program includes a subroutine that makes a reversed
```

```
copy of a
 * list of ints.  The main program simply tests that
routine on a few lists.
 */
public class ReverseListDemo {


   /**
    * Objects of type ListNode are linked together into
linked lists.
    */
   static class ListNode {
      int item;        // An item in the list.
      ListNode next;   // Pointer to the next node in the
list.
   }


   /**
    * Return a new list containing the same items as the
list,
    * but in the reverse order.
    */
   static ListNode reverse( ListNode list ) {
      ListNode rev = null;     // rev will be the reversed
list.
      ListNode runner = list;  // For running through the
nodes of list.
      while (runner != null) {
             // "Push" the next node of list onto the front
of rev.
         ListNode newNode = new ListNode();
         newNode.item = runner.item;
         newNode.next = rev;
         rev = newNode;
            // Move on to the next node in the list.
         runner = runner.next;
      }
      return rev;
   } // end reverse()


   /**
    * Prints the items in the list to which the parameter,
start, points.
    * They are printed on one line, separated by spaces and
enclosed in
    * parentheses.
```

```
    */
   static void printList(ListNode start) {
       ListNode runner;  // For running along the list.
       runner = start;
       System.out.print("(");
       while (runner != null) {
          System.out.print(" " + runner.item);
          runner = runner.next;
       }
       System.out.print(" )");
   } // end printList()


   public static void main(String[] args) {

      System.out.println("I will print out a list and its
reverse, for");
      System.out.println("several different lists.  The
first list is empty.\n");

      ListNode list = null;   // A list, initially empty.
      ListNode reversedList;  // The reversed list.

      int ct = 0;  // How many lists have we processed?

      while (true) {
             // Print the current list and its reverse.
Then
             // add a new node onto the head of the list
before
             // repeating.
          System.out.print("The list:          ");
          printList(list);
          System.out.println();
          reversedList = reverse(list);
          System.out.print("The reversed list: ");
          printList(reversedList);
          System.out.println();
          System.out.println();
          ct++;
          if (ct == 6)
             break;
          ListNode head = new ListNode();  // A new node to
add to the list.
          head.item = (int)(Math.random()*100);  // A
random item.
          head.next = list;
          list = head;
```

```
        }

    } // end main()

} // end class ReverseListDemo
```

Solution for Programming Exercise 9.4

Exercise 9.4:

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.

Discussion

There's really not a lot to think about here, since such a complete algorithm is given. However, we do have to assemble the pieces. I use the standard binary tree node from Section 9.4 (except that I changed the name of the tree node class to *StrTreeNode*). The algorithm needs a queue of tree nodes. To implement this, I copied the *QueueOfInts* class from Subsection 9.3.2 and changed the type of the items in the queue to *StrTreeNode*. I also changed the name to *TreeQueue*. I did this literally by copying the class from a Web

browser window and pasting it into my source code file. With these classes in hand, the algorithm given in the exercise can be coded as:

```
/**
 * Use a queue to print all the strings in the tree to
which
 * root points.  (The nodes will be listed in "level
order",
 * that is:  first the root, then children of the root,
then
 * grandchildren of the root, and so on.)
 */
static void levelOrderPrint(StrTreeNode root) {
    if (root == null)
       return;  // There is nothing to print in an empty
tree.
    TreeQueue queue;   // The queue.
    queue = new TreeQueue();
    queue.enqueue(root);
    while ( queue.isEmpty() == false ) {
       StrTreeNode node = queue.dequeue();
       System.out.println( node.item );
       if ( node.left != null )
          queue.enqueue( node.left );
       if ( node.right != null )
          queue.enqueue( node.right );
    }
} // end levelOrderPrint()
```

The name of this routine comes from the order in which it prints out the nodes of the tree. Think of the root of the tree as being on the top "level" of the tree, the children of the root on the second level, the children of the children of the root on the third level, and so on. Then the subroutine prints the items in **level order**. That is, all the nodes on one level are printed before any of the nodes on the next level. This is a consequence of the way the algorithm processes the items. As items from one level are removed from the queue and printed, their children (which are the nodes on the next level) are added to the **back** of the queue. Just after all the items from one level have been processed, the queue contains all the children of those items, ready to be processed, and those children are exactly the nodes on the next level of the tree. Level-order tree traversals can't be done by recursion, and they are a standard application of queues.

To test my subroutine, I wanted a reasonably large tree whose structure I knew, so I could check whether the nodes are printed in the correct order. (Since you didn't know about level-order traversals until now, on the other hand, you should have been mainly concerned with checking that all the nodes in the tree are printed, period.) I decided to create the binary sort tree shown in Subsection 9.4.2. To do this, I copied the `treeInsert()` subroutine from that section and used it to add names to the tree in an order that would produce the tree I wanted. Finally, I called `levelOrderPrint()` to output the names from the tree. (It worked!)

By the way, you might notice that the `levelOrderPrint()` routine is very similar to the technique used in the grid-marking algorithm in the sample program DepthBreadth.java from Subsection 9.3.2. In fact they are just variations on the same idea. One difference is that in DepthBreadth.java, the squares of the grid had to be marked as "visited" as they were processed to avoid going into an infinite loop. The `levelOrderPrint()` subroutine doesn't have to do the same type of marking because it is working on a tree. One of the defining properties of a tree is that it cannot contain a loop of nodes. That is, it is not possible for a node to be its own descendant. This restriction guarantees that `levelOrderPrint()` will not go into an infinite loop. The same property guarantees that all of our recursive tree-processing methods will not suffer from infinite recursion when they are applied to a tree. You should note, however, that it is possible to connect tree nodes into data structures that contain loops and are therefore not trees at all. While these data structures are not trees, they might have other uses. Many of the subroutines we've looked will fail if applied to these loopy structures.

The Solution

```
/**
 * This program includes a non-recursive subroutine that
prints the
 * nodes of a binary tree, using a queue.  The main program
simply
 * tests that routine.  (The nodes are printed in what is
called
 * "level order".)
 *
```

```
 * This file defines the queue and tree classes as nested
classes.
 * Since they are general-purpose classes, it would really
be better
 * to put them in separate files.
 */


public class TreePrintNonRecursive {


   //---------------------------------- NESTED CLASSES -----
------------------

   /**
    * An object in this class is a node in a binary tree
    * in which the nodes contain items of type String.
    */
   static class StrTreeNode {
      String item;  // The item
      StrTreeNode left;  // Pointer to left subtree.
      StrTreeNode right; // Pointer to right subtree.
      StrTreeNode( String str ) {
            // Constructor.  Make a node to contain str.
         item = str;
      }
   } // end class StrTreeNode


   /**
    * An object of this type represents a queue of
StrTreeNodes,
    * with the usual operations: dequeue, enqueue, isEmpty.
    */
   static class TreeQueue {

      /**
       * An object of type Node holds one of the items
       * in the linked list that represents the queue.
       */
      private static class Node {
         StrTreeNode 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.
       */
      void enqueue( StrTreeNode tree ) {
         Node newTail = new Node();  // A Node to hold the
new item.
         newTail.item = tree;
         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.
       */
      StrTreeNode dequeue() {
         if ( head == null)
             throw new IllegalStateException("Can't dequeue
from an empty queue.");
         StrTreeNode 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, false if
contains one
       * or more items
       */
      boolean isEmpty() {
         return (head == null);
      }

   } // end class TreeQueue


   //-------------------- END OF NESTED CLASSES -----------
----------------


   static StrTreeNode root;  // A pointer to the root of
the binary tree.


   /**
    * Use a queue to print all the strings in the tree to
which
    * root points.  (The nodes will be listed in "level
order",
    * that is:  first the root, then children of the root,
then
    * grandchildren of the root, and so on.)
    */
   static void levelOrderPrint(StrTreeNode root) {
       if (root == null)
          return;  // There is nothing to print in an empty
tree.
       TreeQueue queue;   // The queue, which will only
hold non-null nodes.
       queue = new TreeQueue();
       queue.enqueue(root);
       while ( queue.isEmpty() == false ) {
          StrTreeNode node = queue.dequeue();
```

```
          System.out.println( node.item );
          if ( node.left != null )
             queue.enqueue( node.left );
          if ( node.right != null )
             queue.enqueue( node.right );
       }
   } // end levelOrderPrint()


   /**
    * Add the word to the binary sort tree to which the
    * global variable "root" refers.  I will use this
    * routine only to create the sample tree on which
    * I will test levelOrderPrint().
    */
   static void treeInsert(String newItem) {
      if ( root == null ) {
              // The tree is empty.  Set root to point to a
new node
              // containing the new item.
          root = new StrTreeNode( newItem );
          return;
       }
       StrTreeNode 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 node there.
                   // Otherwise, advance runner down one
level to the left.
             if ( runner.left == null ) {
                runner.left = new StrTreeNode( 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 StrTreeNode( newItem );
                return;  // New item has been added to the
tree.
             }
             else
                runner = runner.right;
          }
       } // end while
   }  // end treeInsert()


   /**
    * Make a tree with a known form, then call
levelOrderPrint()
    * for that tree.  (I want to check that all the items
from
    * the tree are printed, and I want to see the order in
which
    * they are printed.  The expected order of output is
    * judy bill mary alice fred tom dave jane joe.  The
    * tree that is built here is from an illustration in
    * Section 9.4.)
    */
   public static void main(String[] args) {
      treeInsert("judy");
      treeInsert("bill");
      treeInsert("fred");
      treeInsert("mary");
      treeInsert("dave");
      treeInsert("jane");
      treeInsert("alice");
      treeInsert("joe");
      treeInsert("tom");
      levelOrderPrint(root);
   } // end main()


} // end class TreePrintNonRecursive
```

Solution for Programming Exercise 9.5

Exercise 9.5:

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.

Discussion

To create the tree, I copied the *TreeNode* class and the *insertTree()* subroutine from Subsection 9.4.2, and I changed the type of the items in the tree from *String* to double. The main program uses a `for` loop to add 1023 random real numbers to the tree:

```
for (int i = 0; i < 1023; i++)
    treeInsert(Math.random());
```

After that, it's just a matter of writing the routines described in the exercise and calling them to get the desired statistics.

A routine for counting the leaves in the tree is similar to the `countNodes()` routine from Subsection 9.4.2. That routine, however, counts every node in the tree and now we only want to count the leaves. A leaf is defined to be a node in which both the `left` and `right` pointers are `null`. In the recursion, one of the base cases is when we come to a tree that consists of nothing but a leaf. In that case, the number of leaves is 1. If the node is not a leaf, then we have to count the number of leaves in each of its subtrees and add the results:

```
/**
 * Return the number of leaves in the tree to which node
points.
 */
static int countLeaves(TreeNode node) {
    if (node == null)
       return 0;  // An empty tree has no leaves.
    else if (node.left == null && node.right == null)
       return 1;  // Node is a leaf.
    else
       return countLeaves(node.left) +
countLeaves(node.right);
} // end countNodes()
```

In general structure, the other two routines are similar. That is, there are two base cases: an empty tree and a tree consisting just of a leaf. In the remaining case—a node that has one or both subtrees non-empty—the routine has to be applied recursively to the subtrees of the node. Look, for example, at the routine for finding the sum of the depths of all the leaves in the tree:

```
/**
 * When called as sumOfLeafDepths(root,0), this will
compute the
 * sum of the depths of all the leaves in the tree to
which root
 * points.  When called recursively, the depth parameter
gives
 * the depth of the node, and the routine returns the sum
of the
 * depths of the leaves in the subtree to which node
points.
 * In each recursive call to this routine, depth goes up
by one.
 */
static int sumOfLeafDepths( TreeNode node, int depth ) {
    if ( node == null ) {
          // Since the tree is empty and there are no
leaves,
          // the sum is zero.
       return 0;
    }
    else if ( node.left == null && node.right == null) {
          // The node is a leaf, and there are no subtrees
of node, so
          // the sum of the leaf depths is just the depth
of this node.
       return depth;
    }
    else {
          // The node is not a leaf.  Return the sum of
the
          // the depths of the leaves in the subtrees.
       return sumOfLeafDepths(node.left, depth + 1)
                   + sumOfLeafDepths(node.right, depth +
1);
    }
} // end sumOfLeafDepth()
```

The most interesting aspect of this routine is the way it uses its `depth` parameter, which is used to keep track of the depth of the `node` in the complete tree (not just the subtree to which `node` points). For the `root`, the depth is 0. Each time the subroutine is called recursively, the `node` is one level deeper in the tree, and the `depth` parameter is correspondingly increased by 1. When we get down to a leaf node, where `node.left` and `node.right` are `null`, the value of `depth` is the depth of that node in the

original tree, and the sum of the depths of the leaves in the subtree, which consists of just this one leaf node, is `depth`. When `node` is not a leaf, the sums for the two subtrees of `node` are computed recursively and are added together to give the sum for all the leaves in the whole subtree to which `node` refers. (If you have trouble believing that this works, remember that recursion works if it works for the base cases and if it correctly breaks down big problems into smaller problems. You don't have to follow the details.)

The routine for computing the maximum depth is similar.

When I ran my program several times, I found that the average depth of the leaves in the tree tended to be about 12—higher than I expected but still only 1/3 more than the average depth in a perfectly balanced tree. The height of the tree tended to be about 20. (The **height** of a tree is defined to be the maximum depth of any node in the tree.)

The Solution

```
/**
  * This program makes a random binary sort tree containing
1023 random
  * real numbers.  It then computes the height of the tree
and the
  * average depth of the leaves of the tree.  Hopefully,
the average
  * depth will tend to be close to 9, which is what it
would be
  * if the tree were perfectly balanced.  The height of the
tree,
  * which is the same as the maximum depth of any leaf, can
be
  * significantly larger.
  */
public class RandomSortTree {


   static TreeNode root;   // Pointer to the binary sort
tree.


   /**
    * An object of type TreeNode represents one node in a
```

```
binary tree of real numbers.
    */
   static class TreeNode {
       double item;      // The data in this node.
       TreeNode left;    // Pointer to left subtree.
       TreeNode right;   // Pointer to right subtree.
       TreeNode(double x) {
              // Constructor.  Make a node containing x.
          item = x;
       }
   } // end class TreeNode


   /**
    * Add x to the binary sort tree to which the global
variable "root" refers.
    */
   static void treeInsert(double x) {
      if ( root == null ) {
              // The tree is empty.  Set root to point to a
new node
              // containing the new item.
          root = new TreeNode( x );
          return;
       }
       TreeNode runner; // Runs down the tree to find a
place for newItem.
       runner = root;   // Start at the root.
       while (true) {
          if ( x < runner.item ) {
                   // 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 node there.
                   // Otherwise, advance runner down one
level to the left.
             if ( runner.left == null ) {
                runner.left = new TreeNode( x );
                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( x );
                return;  // New item has been added to the
tree.
             }
             else
                runner = runner.right;
           }
       } // end while
   }  // end treeInsert()


   /**
    * Return the number of leaves in the tree to which node
points.
    */
   static int countLeaves(TreeNode node) {
       if (node == null)
          return 0;
       else if (node.left == null && node.right == null)
          return 1;  // Node is a leaf.
       else
          return countLeaves(node.left) +
countLeaves(node.right);
   } // end countNodes()


   /**
    * When called as sumOfLeafDepths(root,0), this will
compute the
    * sum of the depths of all the leaves in the tree to
which root
    * points.  When called recursively, the depth parameter
gives
    * the depth of the node, and the routine returns the
sum of the
    * depths of the leaves in the subtree to which node
points.
    * In each recursive call to this routine, depth goes up
by one.
```

```
    */
   static int sumOfLeafDepths( TreeNode node, int depth ) {
       if ( node == null ) {
             // Since the tree is empty and there are no
leaves,
             // the sum is zero.
          return 0;
       }
       else if ( node.left == null && node.right == null) {
             // The node is a leaf, and there are no
subtrees of node, so
             // the sum of the leaf depth is just the
depths of this node.
          return depth;
       }
       else {
             // The node is not a leaf.  Return the sum of
the
             // the depths of the leaves in the subtrees.
          return sumOfLeafDepths(node.left, depth + 1)
                      + sumOfLeafDepths(node.right, depth +
1);
       }
   } // end sumOfLeafDepths()


   /**
    * When called as maximumLeafDepth(root,0), this will
compute the
    * max of the depths of all the leaves in the tree to
which root
    * points.  When called recursively, the depth parameter
gives
    * the depth of the node, and the routine returns the
max of the
    * depths of the leaves in the subtree to which node
points.
    * In each recursive call to this routine, depth goes up
by one.
    */
   static int maximumLeafDepth( TreeNode node, int depth )
{
       if ( node == null ) {
            // The tree is empty.  Return 0.
          return 0;
       }
       else if ( node.left == null && node.right == null) {
             // The node is a leaf, so the maximum depth in
```

```
this
            // subtree is the depth of this node (the only
leaf
            // that it contains).
         return depth;
      }
      else {
            // Get the maximum depths for the two subtrees
of this
            // node.  Return the larger of the two values,
which
            // represents the maximum in the tree overall.
         int leftMax = maximumLeafDepth(node.left, depth +
1);
         int rightMax =  maximumLeafDepth(node.right,
depth + 1);
         if (leftMax > rightMax)
            return leftMax;
         else
            return rightMax;
      }
   } // end maximumLeafDepth()


   /**
    * The main routine makes the random tree and prints the
statistics.
    */
   public static void main(String[] args) {

      root = null;  // Start with an empty tree.  Root is a
global
                    // variable, defined at the top of the
class.

      // Insert 1023 random items.

      for (int i = 0; i < 1023; i++)
          treeInsert(Math.random());

      // Get the statistics.

      int leafCount = countLeaves(root);
      int depthSum = sumOfLeafDepths(root,0);
      int depthMax = maximumLeafDepth(root,0);
      double averageDepth = ((double)depthSum) / leafCount;

      // Display the results.
```

```
      System.out.println("Number of leaves:         " +
leafCount);
      System.out.println("Average depth of leaves:  " +
averageDepth);
      System.out.println("Maximum depth of leaves:  " +
depthMax);

   }  // end main()


} // end class RandomSortTree
```

Solution for Programming Exercise 9.6

Exercise 9.6:

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.

Discussion

Like the other expression node classes, the *VariableNode* class is a subclass of *ExpNode*, and it must implement the `value(x)` and `printStackCommands()` methods that it inherits from that class. The `value(x)` method has been modified to have a parameter of type double, which gives the value of the variable x. Since a *VariableNode* represents an occurrence of the variable x, the value of the node is simply the value of x. As for the stack commands to evaluate the node: When we encounter an x in an expression, we need to push the value of x onto the stack, just as we would push a constant value onto the stack. I represent this with a stack operation "Push X". Note that x can have different values at different times, so we can't say what value will be pushed. We are generating instructions for a stack machine. At the time when the stack machine **evaluates** the expression, it has to know the value of x. The "Push X" command tells it to push a copy of that value onto the stack. The *VariableNode* class is defined as:

```
/**
 * An expression node that represents a reference to the
variable, x.
 */
private static class VariableNode extends ExpNode {
   VariableNode() {
          // Construct a VariableNode. (There is nothing
to do!)
   }
   double value(double xValue) {
         // The value of the node is the value of x.
      return xValue;
   }
   void printStackCommands() {
         // On a stack machine, just push the value of X
onto the stack.
      System.out.println("  Push X");
   }
}
```

One curious thing about this class is that it doesn't have any instance variables. A *VariableNode* represents an occurrence of x. There is no other

information to record. It's not like a *ConstNode* where we need an instance variable to tell us **which** numerical constant has been found. There is only one "x". Of course, if we expanded our definition of expression to allow other variables such as y and z, we might add an instance variable to *VariableNode* to say which of the possible variables is represented.

The `factorTree()` subroutine from SimpleParser3.java has to be modified to check for an x. If it finds one, it has to return a new *VariableNode*. This change and the others you have to make are fairly straightforward. They are shown in red italic in the solution that follows.

The Solution

```
import textio.TextIO;

/*
    This program reads standard expressions typed in by the
user.
    The program constructs an expression tree to represent
the
    expression.  It then prints the value of the tree.  It
also uses
    the tree to print out a list of commands that could be
used
    on a stack machine to evaluate the expression.
    The expressions can use the variable "x", positive real

numbers, and
    the binary operators +, -, *, and /.  The unary minus
operation
    is supported.  The expressions are defined by the BNF
rules:

            <expression>  ::=  [ "-" ] <term> [ [ "+" | "-"
] <term> ]...

            <term>  ::=  <factor> [ [ "*" | "/" ] <factor>
]...

            <factor>  ::=  <number>  |  <x-variable> | "("
<expression> ")"

    A number must begin with a digit (i.e., not a decimal
```

```
point).
    A line of input must contain exactly one such
expression.  If extra
    data is found on a line after an expression has been
read, it is
    considered an error.

    In addition to the main program class, SimpleParser4,
this program
    defines a set of five nested classes for implementing
expression trees.

 */

public class SimpleParser4 {

//   -------------------- Nested classes for Expression
Trees ------------------------------


   /**
    *  An abstract class representing any node in an
expression tree.
    *  The four concrete node classes are concrete
subclasses.
    *  Two instance methods are specified, so that they can
be used with
    *  any ExpNode.  The value() method returns the value
of the
    *  expression for a specified value of the variable, x.
    *  The printStackCommands() method prints a list

    *  of commands that could be used to evaluate the
expression on
    *  a stack machine (assuming that the value of the
expression is
    *  to be left on the stack).
    */
   abstract private static class ExpNode {
      abstract double value(double xValue);
      abstract void printStackCommands();
   }

   /**
    * Represents an expression node that holds a number.
    */
   private static class ConstNode extends ExpNode {
      double number;  // The number.
```

```
      ConstNode(double val) {
             // Construct a ConstNode containing the
specified number.
         number = val;
      }
      double value(double xValue) {
             // The value of the node is the number that it
contains.
         return number;
      }
      void printStackCommands() {
             // On a stack machine, just push the number
onto the stack.
         System.out.println("  Push " + number);
      }
   }


   /**
    * An expression node representing a binary operator,
    */
   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(double xValue) {
             // The value is obtained by evaluating the
left and right
             // operands and combining the values with the
operator.
         double x = left.value(xValue);
         double y = right.value(xValue);
         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();
         System.out.println("  Operator " + op);
      }
   }


   /**
    * An expression node to represent a unary minus
operator.
    */
   private static class UnaryMinusNode extends ExpNode {
      ExpNode operand;  // The operand to which the unary
minus applies.
      UnaryMinusNode(ExpNode operand) {

             // Construct a UnaryMinusNode with the
specified operand.
         assert operand != null;
         this.operand = operand;
      }
      double value(double xValue) {
             // The value is the negative of the value of
the operand.
         double neg = operand.value(xValue);
         return -neg;
      }
      void printStackCommands() {
             // To evaluate this expression on a stack
machine, first do
             // whatever is necessary to evaluate the
operand, leaving the
```

```
             // operand on the stack.  Then apply the unary
minus (which means
             // popping the operand, negating it, and
pushing the result).
         operand.printStackCommands();
         System.out.println("  Unary minus");
      }
   }


   /**
    * An expression node that represents a reference to the
variable, x.
    */
   private static class VariableNode extends ExpNode {
      VariableNode() {
             // Construct a VariableNode. (There is nothing
to do!)
      }
      double value(double xValue) {
            // The value of the node is the value of x.
         return xValue;
      }
      void printStackCommands() {
            // On a stack machine, just push the value of X
onto the stack.
         System.out.println("  Push X");
      }
   }


//   ---------------------------------------------------------
-------------------------


   /**
    * 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
```

```
   public static void main(String[] args) {

      while (true) {
         System.out.println("\n\nEnter an expression, or
press return to end.");
         System.out.print("\n?  ");
         TextIO.skipBlanks();
         if ( TextIO.peek() == '\n' )
            break;
         try {
            ExpNode exp = expressionTree();
            TextIO.skipBlanks();
            if ( TextIO.peek() != '\n' )
               throw new ParseError("Extra data after end
of expression.");
            TextIO.getln();
            System.out.println("\nValue at x = 0 is " +
exp.value(0));
            System.out.println("Value at x = 1 is " +
exp.value(1));
            System.out.println("Value at x = 2 is " +
exp.value(2));
            System.out.println("Value at x = 3 is " +
exp.value(3));
            System.out.println("\nOrder of postfix
evaluation is:\n");
            exp.printStackCommands();
         }
         catch (ParseError e) {
            System.out.println("\n*** Error in input:    "
+ e.getMessage());
            System.out.println("*** Discarding input:  " +
TextIO.getln());
         }
      }

      System.out.println("\n\nDone.");

   } // end main()


   /**
    * Reads an expression from the current line of input
and builds
    * an expression tree that represents the expression.
    * @return an ExpNode which is a pointer to the root
```

```
node of the
    *    expression tree
    * @throws ParseError if a syntax error is found in the
input
    */
   private static ExpNode expressionTree() throws
ParseError {
      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 the first term.
      if (negative)
         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();
         exp = new BinOpNode(op, exp, nextTerm);
         TextIO.skipBlanks();
      }
      return exp;
   } // end expressionTree()


   /**
    * Reads a term from the current line of input and
builds
    * an expression tree that represents the expression.
    * @return an ExpNode which is a pointer to the root
node of the
    *    expression tree
    * @throws ParseError if a syntax error is found in the
input
    */
   private static ExpNode termTree() throws ParseError {
      TextIO.skipBlanks();
      ExpNode term;  // The expression tree representing
```

```
the term.
      term = factorTree();
      TextIO.skipBlanks();
      while ( TextIO.peek() == '*' || TextIO.peek() == '/'
) {
            // Read the next factor, and combine it with
the
            // previous factors into a bigger expression
tree.
         char op = TextIO.getAnyChar();
         ExpNode nextFactor = factorTree();
         term = new BinOpNode(op,term,nextFactor);
         TextIO.skipBlanks();
      }
      return term;
   } // end termValue()


   /**
    * Reads a factor from the current line of input and
builds
    * an expression tree that represents the expression.
    * @return an ExpNode which is a pointer to the root
node of the
    *    expression tree
    * @throws ParseError if a syntax error is found in the
input
    */

   private static ExpNode factorTree() throws ParseError {
      TextIO.skipBlanks();
      char ch = TextIO.peek();
      if ( Character.isDigit(ch) ) {
            // The factor is a number.  Return a
ConstNode.
         double num = TextIO.getDouble();
         return new ConstNode(num);
      }
      else if ( ch == 'x' || ch == 'X' ) {
            // The factor is the variable x.
         TextIO.getAnyChar();   // Read the X.
         return new VariableNode();
      }
      else if ( ch == '(' ) {
            // The factor is an expression in parentheses.
            // Return a tree representing that expression.
         TextIO.getAnyChar();  // Read the "("
         ExpNode exp = expressionTree();
```

```
         TextIO.skipBlanks();
         if ( TextIO.peek() != ')' )
            throw new ParseError("Missing right
parenthesis.");
         TextIO.getAnyChar();  // Read the ")"
         return exp;
      }
      else if ( ch == '\n' )
         throw new ParseError("End-of-line encountered in
the middle of an expression.");
      else if ( ch == ')' )
         throw new ParseError("Extra right parenthesis.");
      else if ( ch == '+' || ch == '-' || ch == '*' || ch
== '/' )
         throw new ParseError("Misplaced operator.");
      else
         throw new ParseError("Unexpected character \"" +
ch + "\" encountered.");
   }  // end factorTree()


} // end class SimpleParser4
```

Solution for Programming Exercise 9.7

Exercise 9.7:

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**.)

Discussion

The solution to Exercise 9.6 already allows the variable x to occur in expressions. Since we are building on that solution, no changes are needed in the parsing routines. There are a few easy changes in the `main()` routine, since it must take the derivative of the expression entered by the user and then work with that derivative. The changes are shown in the solution that is given below.

Aside from that, we only need to add the "`ExpNode derivative()`" method to each of the node classes. Since I want to print out the derivative in fully parenthesized infix form, I also add another method, "`void printInfix()`". Since this is not a required part of the exercise—and since it's fairly simple to do—I won't discuss the `printInfix()` method further.

All the information that is needed for writing the `derivative()` methods is given in the derivative rules that are listed in the exercise. The first three

rules are pretty simple:

- Since the derivative of a constant is 0, the `derivative()` method in the *ConstNode* class has to return an *ExpNode* that represents the expression "0". That's easy. We just need a constant node that contains the number 0. The definition of `derivative()` in the *ConstNode* class is just: "`return new ConstNode(0);`".
- Similarly, the derivative of x is 1, so the definition of `derivative()` in the *VariableNode* class is "`return new ConstNode(1);`".
- The derivative of `-A` is `-dA`, that is, it consists of a unary minus operator applied to the derivative of the operand A. So, in the *UnaryMinusNode* class, we have to compute the derivative of the `operand` and then create an *ExpNode* that applies a unary minus to that derivative. The derivative of `operand` is `operand.derivative()`, so we only need to "`return new UnaryMinusNode(operand.derivative());`".

In the *BinOpNode* class, the derivative rule that we need to apply depends on the value of the binary operator, `+`, `-`, `*`, or `/`. The rules for `A+B` and `A-B` are easy to implement. Let's look at the case of `*`, where the rule is that the derivative of `A*B` is `A*dB+B*dA`. In the *BinOpNode* class, `A` is the `left` operand and `B` is the right operand. We can compute the derivatives `dA` and `dB` as `left.derivative()` and `right.derivative()`. We then have to build an expression tree to represent `A*dB+B*dA`. We need one node to represent the `+` operation and two more nodes to represent the two `*` operations. We can create the tree step-by-step:

```
ExpNode dA = left.derivative();
ExpNode dB = right.derivative();
ExpNode firstHalf = new BinOpNode('*', left, dB);   //
A*dB
ExpNode secondHalf = new BinOpNode('*', right, dA); //
B*dA
ExpNode answer = new BinOpNode('+', firstHalf,
secondHalf);
return answer;  // This is the derivative we want!
```

In my solution, however, I did the same thing in one statement:

```
return new BinOpNode( '+',
                   new BinOpNode('*', left,
```

```
right.derivative()),
                    new BinOpNode('*', right,
left.derivative()) );
```

This uses the fact that a constructor call is an expression and can be used as an actual parameter in a subroutine. This statement returns a node that represents the sum of two things. The first thing is a node that represents the product of `left` and `right.derivative()`, and the second is a node that represents the product of `right` and `left.derivative()`. This is exactly the same thing that is returned by the previous sequence of six statements. There are reasonable arguments for doing things either way.

The rule for `A/B` is even more complicated: `(B*dA-A*dB)/(B*B)`. Nevertheless, using `left` for `A` and `right` for `B`, I can compute the value with a single statement:

```
return new BinOpNode( '/',
            new BinOpNode('-',
                    new BinOpNode('*', right,
left.derivative()),
                    new BinOpNode('*', left,
right.derivative())),
            new BinOpNode('*', right, right) );
```

As an exercise, you might try doing the same thing with a sequence of simple statements.

The Solution

```
import textio.TextIO;

/*
    This program reads standard expressions typed in by the
user.
    The program constructs an expression tree to represent
the
    expression.  It computes the derivative of the
expression and
    prints out the derivative and the value of the
derivative at
    several values of x.  It also prints out a list of
commands
```

```
    that could be used on a stack machine to evaluate the
derivative.
    The expressions can use the variable "x", positive real
numbers, and
    the binary operators +, -, *, and /.  The unary minus
operation
    is supported.  The expressions are defined by the BNF
rules:

            <expression>  ::=  [ "-" ] <term> [ [ "+" | "-"
] <term> ]...

            <term>  ::=  <factor> [ [ "*" | "/" ] <factor>
]...

            <factor>  ::=  <number>  |  <x-variable> | "("
<expression> ")"

    A number must begin with a digit (i.e., not a decimal
point).
    A line of input must contain exactly one such
expression.  If extra
    data is found on a line after an expression has been
read, it is
    considered an error.

    In addition to the main program class, SimpleParser5,
this program
    defines a set of five nested classes for implementing
expression trees.

 */

public class SimpleParser5 {

//   -------------------- Nested classes for Expression
Trees ------------------------------


   /**
    *  An abstract class representing any node in an
expression tree.
    *  The four concrete node classes are concrete
subclasses.
    *  Two instance methods are specified, so that they can
be used with
    *  any ExpNode.  The value() method returns the value
of the
```

```
of the
    *  expression for a specified value of the variable, x.
    *  The printStackCommands() method prints a list
    *  of commands that could be used to evaluate the
expression on
    *  a stack machine (assuming that the value of the
expression is
    *  to be left on the stack).
    *  The derivative() method returns an expression tree
for the derivative
    *  of the expression (with no attempt at
simplification).  Actually,
    *  this might not be a tree, but it is a "directed
acyclic graph",

    *  with no loops, so it's OK for our purposes.  The
printInfix()
    *  method prints the expression in fully parenthesized
form.
    */
   abstract private static class ExpNode {
      abstract double value(double xValue);
      abstract void printStackCommands();
      abstract void printInfix();
      abstract ExpNode derivative();
   }

   /**
    * Represents an expression node that holds a number.
    */
   private static class ConstNode extends ExpNode {
      double number;  // The number.
      ConstNode(double val) {
             // Construct a ConstNode containing the
specified number.
         number = val;
      }
      double value(double xValue) {
             // The value of the node is the number that it
contains.
         return number;
      }
      void printStackCommands() {
             // On a stack machine, just push the number
onto the stack.
         System.out.println("  Push " + number);
      }
      void printInfix() {
```

```
         System.out.print(number);
      }
      ExpNode derivative() {
            // The derivative of a constant is zero.
         return new ConstNode(0);
      }
   }


   /**
    * An expression node representing a binary operator,
    */
   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(double xValue) {
            // The value is obtained by evaluating the
left and right
            // operands and combining the values with the
operator.
         double x = left.value(xValue);
         double y = right.value(xValue);
         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
```

```
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();
         System.out.println("  Operator " + op);
      }
      void printInfix() {
            // Print the expression, in parentheses.
         System.out.print('(');

         left.printInfix();
         System.out.print(" " + op + " ");
         right.printInfix();
         System.out.print(')');
      }
      ExpNode derivative() {
            // Apply the derivative formulas.
         switch (op) {
         case '+':
            return new BinOpNode('+', left.derivative(),
right.derivative());
         case '-':
            return new BinOpNode('-', left.derivative(),
right.derivative());
         case '*':
            return new BinOpNode( '+',
                  new BinOpNode('*', left,
right.derivative()),
                  new BinOpNode('*', right,
left.derivative()) );
         case '/':
            return new BinOpNode( '/',
                  new BinOpNode('-',
                        new BinOpNode('*', right,
left.derivative()),
                        new BinOpNode('*', left,
right.derivative())),
                        new BinOpNode('*', right, right) );
         default:
            return null;
         }
      }
```

```
   }


   /**
    * An expression node to represent a unary minus
operator.
    */
   private static class UnaryMinusNode extends ExpNode {
      ExpNode operand;  // The operand to which the unary
minus applies.
      UnaryMinusNode(ExpNode operand) {
             // Construct a UnaryMinusNode with the
specified operand.
         assert operand != null;
         this.operand = operand;
      }
      double value(double xValue) {
             // The value is the negative of the value of
the operand.
         double neg = operand.value(xValue);
         return -neg;
      }
      void printStackCommands() {
             // To evaluate this expression on a stack
machine, first do
             // whatever is necessary to evaluate the
operand, leaving the
             // operand on the stack.  Then apply the unary
minus (which means
             // popping the operand, negating it, and
pushing the result).
         operand.printStackCommands();
         System.out.println("  Unary minus");
      }
      void printInfix() {
         // Print the expression, in parentheses.
         System.out.print("(-");
         operand.printInfix();
         System.out.print(')');
      }
      ExpNode derivative() {
         // The derivative of -A is -(derivative of A).
         return new UnaryMinusNode(operand.derivative());
      }
   }
```

```
   /**
    * An expression node that represents a reference to the
variable, x.
    */
   private static class VariableNode extends ExpNode {
      VariableNode() {
             // Construct a VariableNode. (There is nothing
to do!)
      }
      double value(double xValue) {
            // The value of the node is the value of x.
         return xValue;
      }
      void printStackCommands() {
            // On a stack machine, just push the value of X
onto the stack.
         System.out.println("  Push X");
      }
      void printInfix() {
         System.out.print("x");
      }
      ExpNode derivative() {
            // The derivative of x is the constant 1.
         return new ConstNode(1);
      }
   }


   //   ---------------------------------------------------
-----------------------------


   /**
    * 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


   public static void main(String[] args) {

      while (true) {
```

```
         System.out.println("\n\nEnter an expression, or
press return to end.");
         System.out.print("\n?  ");
         TextIO.skipBlanks();
         if ( TextIO.peek() == '\n' )
            break;
         try {
            ExpNode exp = expressionTree();
            TextIO.skipBlanks();
            if ( TextIO.peek() != '\n' )
               throw new ParseError("Extra data after end
of expression.");
            TextIO.getln();
            ExpNode deriv = exp.derivative();

            System.out.println("\nA fully parenthesized
expression for the derivative is:");
            System.out.print("   ");
            deriv.printInfix();
            System.out.println();
            System.out.println("\nValue of derivative at x
= 0 is " + deriv.value(0));
            System.out.println("Value of derivative at x =
1 is " + deriv.value(1));
            System.out.println("Value of derivative at x =
2 is " + deriv.value(2));
            System.out.println("Value of derivative at x =
3 is " + deriv.value(3));
            System.out.println("\nOrder of postfix
evaluation for derivative is:\n");
            deriv.printStackCommands();
         }
         catch (ParseError e) {
            System.out.println("\n*** Error in input:    "
+ e.getMessage());
            System.out.println("*** Discarding input:  " +
TextIO.getln());
         }
      }

      System.out.println("\n\nDone.");

   } // end main()


   /**
    * Reads an expression from the current line of input
and builds
```

```
and builds
    * an expression tree that represents the expression.
    * @return an ExpNode which is a pointer to the root
node of the
    *    expression tree
    * @throws ParseError if a syntax error is found in the
input
    */
   private static ExpNode expressionTree() throws
ParseError {
      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 the first term.
      if (negative)
         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();
         exp = new BinOpNode(op, exp, nextTerm);
         TextIO.skipBlanks();
      }
      return exp;
   } // end expressionTree()


   /**
    * Reads a term from the current line of input and
builds
    * an expression tree that represents the expression.
    * @return an ExpNode which is a pointer to the root
node of the
    *    expression tree
    * @throws ParseError if a syntax error is found in the
input
    */
   private static ExpNode termTree() throws ParseError {
```

```
   private static ExpNode termTree() throws ParseError {
      TextIO.skipBlanks();
      ExpNode term;  // The expression tree representing
the term.
      term = factorTree();
      TextIO.skipBlanks();
      while ( TextIO.peek() == '*' || TextIO.peek() == '/'
) {
            // Read the next factor, and combine it with
the
            // previous factors into a bigger expression
tree.
         char op = TextIO.getAnyChar();
         ExpNode nextFactor = factorTree();
         term = new BinOpNode(op,term,nextFactor);
         TextIO.skipBlanks();
      }
      return term;
   } // end termValue()


   /**
    * Reads a factor from the current line of input and
builds
    * an expression tree that represents the expression.
    * @return an ExpNode which is a pointer to the root
node of the
    *    expression tree
    * @throws ParseError if a syntax error is found in the
input
    */

   private static ExpNode factorTree() throws ParseError {
      TextIO.skipBlanks();
      char ch = TextIO.peek();
      if ( Character.isDigit(ch) ) {
            // The factor is a number.  Return a
ConstNode.
         double num = TextIO.getDouble();
         return new ConstNode(num);
      }
      else if ( ch == 'x' || ch == 'X' ) {
            // The factor is the variable x.
           TextIO.getAnyChar();   // Read the X.
           return new VariableNode();
         }
      else if ( ch == '(' ) {
            // The factor is an expression in parentheses.
            // Return a tree representing that expression
```

```
            // Return a tree representing that expression.
         TextIO.getAnyChar();  // Read the "("
         ExpNode exp = expressionTree();
         TextIO.skipBlanks();
         if ( TextIO.peek() != ')' )
            throw new ParseError("Missing right
parenthesis.");
         TextIO.getAnyChar();  // Read the ")"
         return exp;
      }
      else if ( ch == '\n' )
         throw new ParseError("End-of-line encountered in
the middle of an expression.");
      else if ( ch == ')' )
         throw new ParseError("Extra right parenthesis.");
      else if ( ch == '+' || ch == '-' || ch == '*' || ch
== '/' )
         throw new ParseError("Misplaced operator.");
      else
         throw new ParseError("Unexpected character \"" +
ch + "\" encountered.");
   }  // end factorTree()


} // end class SimpleParser5
```

Solution for Programming Exercise 10.1

Exercise 10.1:

Rewrite the *PhoneDirectory* class from Subsection 7.5.2 so that it uses a *TreeMap* to store directory entries, instead of an array. (Doing this was suggested in Subsection 10.3.1.) You should also write a short program to test the class.

Discussion

The class that you are asked to write for this exercise really is very trivial. Everything that the class needs to do is already done by the *TreeMap* class, and *TreeMap* does much more besides. In fact the only reason to write the *PhoneDirectory* class is to **hide** a lot of the capabilities of *TreeMaps*. A *PhoneDirectory* object is a "thin wrapper" around a *TreeMap* object. It allows only the operations `getNumber(name)` and `putNumber(name,number)`. Each of these can be implemented in a single line, as a call to one of the methods in the *TreeMap*. The *PhoneDirectory* does provide nice, meaningful names for the methods, and it adds a check to the implementation of `putNumber(name,number)` to ensure that `name` and `number` are not `null`.

It would be nice for phone directories to have a few more capabilities. A method for listing the contents of the directory would be especially nice, so I have added one to my class even though there was no such method in the original *PhoneDirectory* class. The print method uses a for-each loop to iterate through the entry set of the phone directory. It is very similar to examples in Subsection 10.3.2 Even with this addition, though, the *PhoneDirectory* class is still very far from being useful in real applications.

The Solution

```
import java.util.Map;
import java.util.TreeMap;

/**
 * A phone directory 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.
 * (This version of the class uses a TreeMap to store the
data.)
 */
public class PhoneDirectoryWithTreeMap {


   /**
    * The TreeMap that will store the data.  Both key and
value are
    * of type String.  The key represents a name and the
value represents
    * the associated phone number.
    */
   private TreeMap<String,String> data;


   /**
    * Constructor creates an initially empty directory.
    */
   public PhoneDirectoryWithTreeMap() {
      data = new TreeMap<String,String>();
   }


   /**
    * 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 ) {
         return data.get(name);
   }


   /**
    * 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");
      data.put(name,number);
   }


   /**
    *  Write the contents of the phone directory to
System.out.
    */
   public void print() {
      for ( Map.Entry<String,String> entry :
data.entrySet() )
         System.out.println( entry.getKey() + ":  " +
entry.getValue() );
   }

} // end class PhoneDirectoryWithTreeMap
```

Here is a very small program to test the class:

```
public class TestPhoneDirectoryWithTreeMap {

   public static void main(String[] args) {
       PhoneDirectoryWithTreeMap directory = new
PhoneDirectoryWithTreeMap();
       System.out.println("This program creates a
PhoneDirectoryWithTreeMap and");
       System.out.println("adds several entries.  It then
prints the contents of");
       System.out.println("directory and does a few
lookups.");
       System.out.println();
       directory.putNumber("Fred","555-1234");
       directory.putNumber("Barney","555-2345");
```

```
        directory.putNumber("Wilma","555-3456");
        System.out.println("Contents are:");
        System.out.println();
        directory.print();
        System.out.println();
        System.out.println("Number for Fred is " +
directory.getNumber("Fred"));
        System.out.println("Number for Wilma is " +
directory.getNumber("Wilma"));
        System.out.println("Number for Tom is " +
directory.getNumber("Tom"));
           // The output from the last line should be null.
    }

}
```

Solution for Programming Exercise 10.2

Exercise 10.2:

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.) For input and output, a set of such integers will be written 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.print(A)`. I've chosen the syntax for sets to be the same as that used by the system for outputting a set.)

Discussion

This program would be **much** easier to write if we could assume that the user's input was in the correct format. Unfortunately, the exercise says that we have to detect errors in the user's input and handle them by printing error messages. To do this we have to constantly look ahead while reading the user's input, to see whether the input agrees with what we are expecting. The techniques for doing this were covered for the example LengthConverter2.java from Subsection 8.2.2. We can use the function `TextIO.peek()` to look ahead at the next character in the user's input. We need to be able to skip past any blanks in the input. Instead of writing my own method to do that, I use the method `TextIO.skipBlanks()`.

My program uses exceptions to handle the errors. Exceptions make it possible to organize the error-handling code in a straightforward way. When an error is discovered, an exception is thrown. The main program uses a `try..catch` statement to try to process one line of input. If an error occurs, the program does not crash. The error is caught, an error message is printed, and the program continues. I throw an error of type *IllegalArgumentException* when an error is found. This exception is a standard part of Java, but it might be better style to define a new error class to represent the specific type of error that occurs in this program.

My program uses a method named `calc()` to read and process one line of input. This method in turn uses another method, `readSet()` to read each of

the two sets of integers from the input line. Without error handling, an algorithm for `readSet()` would be:

```
Start with an empty set.
Read the '[' that begins the set.
Repeat:
   Read the next number and add it to the set.
   If the next character is ']':
      break.
   Read the comma that separates one number from the next.
Read the ']'.
Return the set.
```

To turn this into a robust routine, we just have to check, before reading anything, that the next character is legal. If not, throw an exception. This adds a lot of code, but the nice thing about throwing exceptions is that it doesn't disturb the logical flow of the routine.

There is one possible bug in the algorithm as given. It assumes that there is at least one number in the set. But in mathematics, a set can be empty. I decided to allow for the possibility of empty sets in my program. See the solution, below.

An algorithm for the `calc()` method is even more straightforward. Again, the basic algorithm ignores the possibility of errors:

```
Read the first set, A.
Read the operator.
Read the second set, B.
if the operator is '+'
   A.addAll(B)  // Sets A equal to the union of A and B.
else if the operator is '*'
   A.retainAll(B)  // Sets A to the intersection.
else
   A.removeAll(B)  // Sets A to the difference.
Print A.
```

In the program, an error check has to be added to make sure that there is a legal operator in the correct position. I also add an error check to make sure that there is no extra data on the line.

The Solution

```
import textio.TextIO;
import java.util.TreeSet;


/**
 * This program is a very simple "set calculator" that can
compute
 * the intersection, union, and set difference of two sets
of
 * non-negative integers.  Each line of the user's input
contains two
 * such sets, separated by one of the operators +, *, or -,
standing
 * for union, intersection, and set difference
respectively.  A set
 * must be given in the form of a list of non-negative
integers, separated
 * by commas, and enclosed in square brackets.  For
example:
 * [1, 2, 3] + [4, 3, 10, 0].  Spaces can occur anywhere in
the input.
 * If an error is found in the input, the program will
report it.
 * The program ends when the user inputs an empty line.
 *
 * This program is mainly a demonstration of Sets.
 */


public class SetCalc {

   public static void main(String[] args) {

      System.out.println("This program will compute union,
intersection,");
      System.out.println("and set difference of sets of
integers.");
      System.out.println("");
      System.out.println("");
      System.out.println("Enter set computations (press
return to end):");

      while (true) {
         char ch;
```

```
         System.out.print("\n? ");
         TextIO.skipBlanks();
         if (TextIO.peek() == '\n') {
               // The input line is empty.
               // Exit the loop and end the program.
            break;
         }
         try {
            calc(); // Reads and processes one line of
input.
         }
         catch (IllegalArgumentException e) {
               // An error was found in the input line.
            System.out.println("Error in input: " +
e.getMessage());
         }
         TextIO.getln();  // Read and discard the rest of
the line.
                          // If there was no error, the
only thing that
                          // is discarded is the end-of-
line character.
      }

   } // end main()


   /**
    * Read a line of input, consisting of two sets
separated by
    * an operator.  Perform the operation and output the
value.
    * If any syntax error is found in the input, an
    * IllegalArgumentException is thrown
    */
   private static void calc() {

      TreeSet<Integer> A, B;  // The two sets of integers.
      char op;                // The operator, +, *, or -.

      A = readSet(); // Read the first set.

      TextIO.skipBlanks();
      if (TextIO.peek() != '*' && TextIO.peek() != '+'
                                          && TextIO.peek()
!= '-')
         throw new IllegalArgumentException(
         "Expected *, +, or  - after first set.");
```

```
      op = TextIO.getAnyChar(); // Read the operator.

      B = readSet(); // Read the second set.

      TextIO.skipBlanks();
      if (TextIO.peek() != '\n')
         throw new IllegalArgumentException("Extra
unexpected input.");

      // Perform the operation.  This modifies set A to
represent
      // the answer.  Display the answer by printing out A.
The
      // output format for a set of integers is the same as
the
      // input format used by this program.

      if (op == '+')
         A.addAll(B);     // Union.
      else if (op == '*')
         A.retainAll(B);  // Intersection.
      else
         A.removeAll(B);  // Set difference.

      System.out.print("Value:  " + A);

   } // end calc()


   /**
    * Reads a set of non-negative integers from standard
input,
    * and stores them in a TreeSet that contains objects
belonging
    * to the wrapper class Integer.  The set must be
enclosed
    * between square brackets and must contain a list of
zero or
    * more non-negative integers, separated by commas.
Spaces
    * are allowed anywhere.  If the input is not of the
correct
    * form, an IllegalArgumentException is thrown.
    */
   private static TreeSet<Integer> readSet() {

      TreeSet<Integer> set = new TreeSet<Integer>();  //
The set that will be read.
```

```
      TextIO.skipBlanks();
      if (TextIO.peek() != '[')
         throw new IllegalArgumentException("Expected '['
at start of set.");
      TextIO.getAnyChar(); // Read the '['.

      TextIO.skipBlanks();
      if (TextIO.peek() == ']') {
            // The set has no integers.  This is the empty
set, which
            // is legal.  Return the value.
         TextIO.getAnyChar(); // Read the ']'.
         return set;
      }

      while (true) {
            // Read the next integer and add it to the set.
         TextIO.skipBlanks();
         if (! Character.isDigit(TextIO.peek()))
            throw new IllegalArgumentException("Expected an
integer.");
         int n = TextIO.getInt(); // Read the integer.
         set.add( Integer.valueOf(n) );  // (Could have
just said set.add(n)!)
         TextIO.skipBlanks();
         if (TextIO.peek() == ']')
            break;  // ']' marks the end of the set.
         else if (TextIO.peek() == ',')
            TextIO.getAnyChar(); // Read a comma and
continue.
         else
            throw new IllegalArgumentException("Expected
',' or ']'.");
      }

      TextIO.getAnyChar(); // Read the ']' that ended the
set.

      return set;

   } // end readSet()


} // end class SetCalc
```

Solution for Programming Exercise 10.3

Exercise 10.3:

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 are **not** required to increase the size of the table when it becomes too full.

You should also write a short program to test your solution.

Discussion

Hash tables are discussed in Subsection 10.3.3. A hash table is just an array of linked lists. Each linked list holds all the items in the table that share the same hash code. Initially, all the lists are empty (represented as `null` in the array). We need to be able to add and delete items in the list. Methods for inserting and deleting items can be found in Subsection 9.2.2. In a hash table, the order of items in a particular list doesn't matter, so I simply insert each new item at the beginning of the list. This makes the insert operation fairly simple. Deletion, however, is more complicated, since we need to be able to delete an item no matter where it occurs in a list.

A hash table really contains pairs of items, where each pair consists of a key and an associated value. To implement this, I use linked lists in which each node contains both a key and a value, as well as the pointer to the next node in the list. The end of a list is marked, as usual, by a `null` pointer. The nodes are defined by a static nested class:

```
private static class ListNode {
   String key;
   String value;
   ListNode next;
}
```

The array of linked lists that stores all the data is of type `ListNode[]`. Each item in the array is either `null` to indicate an empty list, or it is a pointer to the first node in a linked list.

Given any key, we can find the linked list that should contain that key by looking at the hash code of the key. The code computed by calling `key.hashCode()`, which returns a value of type int. We need a value that is in the range of legal indices for the array. As noted in Subsection 10.3.3, the value can be computed as `Math.abs(key.hashCode()) % table.length`, where `table` is the array.

The hash code is used in all the methods that deal with keys, to decide which linked list to look at. Once a list has been selected, the operations on the list are pretty straightforward (given all the information and code in Subsection 9.2.2), so I will not discuss them further here. You can look at the solution, below.

Although it is not required by the exercise, I defined a `resize()` method that is used to increase the size of the table when the table becomes too full. I call this method in the `put()` method when the table becomes more than 3/4 full. The hash table class has an instance variable, `count`, that is equal to the number of items in the table, so it only necessary to check the value of `count` when deciding to increase the array size. Having `count` is also important for implementing the `size()` method efficiently, since `size()` can simply return `count` as its return value.

(By the way, the statement that "`count` is equal to the number of items in the table" is a class invariant for the *HashTable* class. (See Subsection 8.2.3.) It is important that all of the methods in the class maintain the truth of this class invariant.")

A major part of developing a class for general use is testing. It's important to design a testing procedure that will test all aspects of the class. For this problem, I wrote a testing program that would allow me to test each of the methods in the class. I also added a method `dump()` to the class; this method displays the entire hash table. This method does not really belong in the class, since users of a hash table shouldn't care how the data is stored. But I needed it to make sure that my `resize()` method is working properly and that I could delete items correctly from all positions in the lists (beginning, middle, and end).

By the way, you might be tempted to try to write a generic version of *HashTable* in which the types of the keys and values are given as type parameters in the definition of the class. Unfortunately, you would run into problems as soon as you tried to use an array in your class. Generic classes don't work well with arrays, because arrays of generic type can't be created at run time, as noted in Subsection 10.1.3.

The Solution

The HashTable Class

```
/**
 * This file defines a HashTable class.  Keys and values in
the table are of
 * type String.  Keys cannot be null.  The default
constructor creates a table
 * that initially has 64 locations, but a different initial
size can be specified
 * as a parameter to the constructor.  The table increases
in size if it
 * becomes more than 3/4 full.
 */
public class HashTable {

   /**
```

```
    * Keys that have the same hash code are placed together
in a linked list.
    * This private nested class is used internally to
implement linked lists.
    * A ListNode holds a (key,value) pair.
    */
   private static class ListNode {
      String key;
      String value;
      ListNode next;  // Pointer to next node in the list;
                      // A null marks the end of the list.
   }

   private ListNode[] table;  // The hash table,

represented as
                              // an array of linked lists.

   private int count;  // The number of (key,value) pairs
in the
                       // hash table.


   /**
    * Create a hash table with an initial array size of 64.
    */
   public HashTable() {
      table = new ListNode[64];
   }


   /**
    * Create a hash table with a specified initial array
size.
    * Precondition: initalSize > 0.
    */
   public HashTable(int initialSize) {
      if (initialSize <= 0)
         throw new IllegalArgumentException("Illegal table
size");
      table = new ListNode[initialSize];
   }


   /**
    * This method is NOT part of the usual interface for a
hash table.
    * It is here only to be used for testing purposes, and
```

```
should be
    * removed before the class is released for general use.
This
    * lists the (key,value) pairs in each location of the
table.
    */
   void dump() {
      System.out.println();
      for (int i = 0; i < table.length; i++) {
             // Print out the location number and the list
of
             // key/value pairs in this location.
         System.out.print(i + ":");
         ListNode list = table[i]; // For traversing linked
list number i.
         while (list != null) {
            System.out.print("  (" + list.key + "," +
list.value + ")");
            list = list.next;
         }
         System.out.println();
      }
   } // end dump()


   /**
    * Associate the specified value with the specified key.
    * Precondition:  The key is not null.
    */
   public void put(String key, String value) {

      assert key != null : "The key must be non-null";

      int bucket = hash(key); // Which location should this
key be in?

      ListNode list = table[bucket]; // For traversing the
linked list
                                     // at the appropriate
location.
      while (list != null) {
            // Search the nodes in the list, to see if the
key already exists.
         if (list.key.equals(key))
            break;
         list = list.next;
      }
```

```
      // At this point, either list is null, or
list.key.equals(key).

      if (list != null) {
            // Since list is not null, we have found the
key.
            // Just change the associated value.
         list.value = value;
      }
      else {
             // Since list == null, the key is not already
in the list.
             // Add a new node at the head of the list to
contain the
             // new key and its associated value.
         if (count >= 0.75*table.length) {
               // The table is becoming too full.  Increase
its size
               // before adding the new node.
            resize();
            bucket = hash(key);  // Recompute hash code,
since it
                                 // depends on the table
size.
         }
         ListNode newNode = new ListNode();
         newNode.key = key;
         newNode.value = value;
         newNode.next = table[bucket];
         table[bucket] = newNode;
         count++;  // Count the newly added key.
      }
   }


   /**
    * Retrieve the value associated with the specified key
in the table,
    * if there is any.  If not, the value null will be
returned.
    * @param key The key whose associated value we want to
find
    * @return the associated value, or null if there is no
associated value
    */
   public String get(String key) {

      int bucket = hash(key);  // At what location should
```

```
the key be?

      ListNode list = table[bucket];  // For traversing the
list.
      while (list != null) {
            // Check if the specified key is in the node
that
            // list points to.  If so, return the
associated value.
         if (list.key.equals(key))
            return list.value;
         list = list.next;  // Move on to next node in the
list.
      }

      // If we get to this point, then we have looked at
every
      // node in the list without finding the key.  Return
      // the value null to indicate that the key is not in
the table.

      return null;
   }


   /**
    * Remove the key and its associated value from the
table,
    * if the key occurs in the table.  If it does not
occur,
    * then nothing is done.
    */
   public void remove(String key) {

      int bucket = hash(key);  // At what location should
the key be?

      if (table[bucket] == null) {
            // There are no keys in that location, so key
does not
            // occur in the table.  There is nothing to do,
so return.
         return;
      }

      if (table[bucket].key.equals(key)) {
            // If the key is the first node on the list,
then
```

```
            // table[bucket] must be changed to eliminate
the
            // first node from the list.
         table[bucket] = table[bucket].next;
         count--; // Remove new number of items in the
table.
         return;
      }

      // We have to remove a node from somewhere in the
middle
      // of the list, or at the end.  Use a pointer to
traverse
      // the list, looking for a node that contains the
      // specified key, and remove it if it is found.

      ListNode prev = table[bucket];  // The node that
precedes
                                      // curr in the list.
Prev.next
                                      // is always equal to
curr.
      ListNode curr = prev.next;  // For traversing the
list,
                                  // starting from the
second node.
      while (curr != null && ! curr.key.equals(key)) {
         curr = curr.next;
         prev = curr;
      }

      // If we get to this point, then either curr is null,
      // or curr.key is equal to key.  In the latter case,
      // we have to remove curr from the list.  Do this
      // by making prev.next point to the node after curr,
      // instead of to curr.  If curr is null, it means
that
      // the key was not found in the table, so there is
nothing
      // to do.

      if (curr != null) {
         prev.next = curr.next;
         count--;  // Record new number of items in the
table.
      }
   }
```

```
   /**
    * Test whether the specified key has an associated
value in the table.
    * @param key The key that we want to search for.
    * @return true if the key exists in the table, false if
not
    */
   public boolean containsKey(String key) {

      int bucket = hash(key);  // In what location should
key be?

      ListNode list = table[bucket];  // For traversing the
list.
      while (list != null) {
            // If we find the key in this node, return
true.
         if (list.key.equals(key))
            return true;
         list = list.next;
      }

      // If we get to this point, we know that the key does
      // not exist in the table.

      return false;
   }


   /**
    * Return the number of key/value pairs in the table.
    */
   public int size() {
      return count;
   }


   /**
    * Compute a hash code for the key; key cannot be null.
    * The hash code depends on the size of the table as
    * well as on the value returned by key.hashCode().
    */
   private int hash(Object key) {
      return (Math.abs(key.hashCode())) % table.length;
   }
```

```
   /**
    * Double the size of the table, and redistribute the
    * key/value pairs to their proper locations in the
    * new table.
    */
   private void resize() {
      ListNode[] newtable = new ListNode[table.length*2];
      for (int i = 0; i < table.length; i++) {
             // Move all the nodes in linked list number i
into the new table.
             // No new ListNodes are created.  The existing
ListNode for each
             // key/value pair is moved to the newtable.
This is done by
             // changing the "next" pointer in the node and
by making a pointer
             // in the new table point to the node.
         ListNode list = table[i]; // For traversing linked
list number i.
         while (list != null) {
               // Move the node pointed to by list to the
new table.
            ListNode next = list.next;  // The is the next
node in the list.
               // Remember it, before changing the value of
list!
            int hash = (Math.abs(list.key.hashCode())) %
newtable.length;
               // hash is the hash code of list.key that is
               // appropriate for the new table size.  The
               // next two lines add the node pointed to by
list
               // onto the head of the linked list in the
new table
               // at position number hash.
            list.next = newtable[hash];
            newtable[hash] = list;
            list = next;  // Move on to the next node in
the OLD table.
         }
      }
      table = newtable;  // Replace the table with the new
table.
   } // end resize()


} // end class HashTable
```

A Class for Testing:

```
import textio.TextIO;

/**
 * A little program to test the HashTable class.  Note that
I
 * start with a really small table so that I can easily
test
 * the resize() method.
 */

public class TestHashTable {

   public static void main(String[] args){
      HashTable table = new HashTable(2);  // Initial size
of table is 2.
      String key,value;
      while (true) {
         System.out.println("\nMenu:");
         System.out.println("   1. test put(key,value)");
         System.out.println("   2. test get(key)");
         System.out.println("   3. test containsKey(key)");
         System.out.println("   4. test remove(key)");
         System.out.println("   5. show complete contents
of hash table.");
         System.out.println("   6. EXIT");
         System.out.print("Enter your command:  ");
         switch ( TextIO.getlnInt()) {
         case 1:
            System.out.print("\n   Key = ");
            key = TextIO.getln();
            System.out.print("   Value = ");
            value = TextIO.getln();
            table.put(key,value);
            break;
         case 2:
            System.out.print("\n   Key = ");
            key = TextIO.getln();
            System.out.println("   Value is " +
table.get(key));
            break;
         case 3:
            System.out.print("\n   Key = ");
            key = TextIO.getln();
```

```
            System.out.println("   containsKey(" + key + ")
is "
                  + table.containsKey(key));
            break;
         case 4:
            System.out.print("\n   Key = ");
            key = TextIO.getln();
            table.remove(key);
            break;
         case 5:
            table.dump();
            break;
         case 6:
            return;  // End program by returning from
main()
         default:
            System.out.println("   Illegal command.");
         break;
         }
         System.out.println("\nHash table size is " +
table.size());
      }
   }

} // end class TestHashTable
```

Solution for Programming Exercise 10.4

Exercise 10.4:

A **predicate** is a boolean-valued function with one parameter. Java has the parameterized functional interface `Predicate<T>`, from package `java.util.function`, to represent predicates. The definition of `Predicate<T>` could be:

```
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. Java already has some methods that use predicates, such as the `removeIf(p)` method that is defined for any *Collection*. (See Subsection 10.6.1). However, this exercise asks you to write a few similar methods yourself. 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. You should **not** use the stream API for this exercise.

```
public static <T> void remove(Collection<T> coll,
Predicate<T> pred)
   // Remove every object, obj, from coll for which
pred.test(obj)
   // is true.  (This does the same thing as
coll.removeIf(pred).)

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.
```

Discussion

The code for this exercise is pretty short, but it involves some advanced ideas, including writing generic methods (from Subsection 10.5.2). Aside from that, the main point of the exercise is to work with predicates and to see how they are useful in generic programming. The operation of finding all the items in a collection that satisfy some condition is very common. The code to do it is not hard to write, but even so, it is nice to have it done once and for all. In this exercise, the condition that the items have to satisfy is represented as a *Predicate*. The `collect()` method provides a general way to find all the items that satisfy the condition. The items, in this case, are to be dumped into a *List*.

The implementation of the `collect()` method simply uses a for-each loop to go through the collection, `coll`, and puts every item that satisfies the predicate into a list. The method returns an object of type *List*, but *List* is only an interface. Any real list must belong to one of the classes that implement this interface, such as *LinkedList* or *ArrayList*. I create the list as an *ArrayList*, but a *LinkedList* would also be reasonable.

```
/**
 * Return a List that contains all the objects, obj, from
the collection,
 * coll, for which pred.test(obj) is true.
 */
public static <T> List<T> collect(Collection<T> coll,
Predicate<T> pred) {
   List<T> list = new ArrayList<T>();
   for ( T item : coll ) {
      if (pred.test(item))
         list.add(item);
   }
```

```
    return list;
} // end collect()
```

The `remove()` and `retain()` methods are similar to `collect()`, except that instead of making a new collection, they modify the collection, `coll`. A for-each loop cannot modify the collection over which it is iterating, so the `remove()` and `retain()` methods will have to use *Iterators*. Recall that if `iter` is an *Iterator* for a collection, then the method `iter.remove()` will remove from the collection the item that was seen most recently by the iterator. The `remove()` method can be written as:

```
/**
 * Remove every object, obj, from coll for which
pred.test(obj) is true.
 */
public static <T> void remove(Collection<T> coll,
Predicate<T> pred) {
   Iterator<T> iter = coll.iterator();
   while (iter.hasNext()) {
      T item = iter.next();
      if (pred.test(item))
         iter.remove();
   }
} // end remove()
```

The `retain()` method is almost identical to this, and the `find()` method is easy to write using an ordinary `for` loop. See my solution, below.

If you look at the documentation for `Predicate<T>`, you will see that it defines a method `negate()`. (It is defined in the interface as a `default` method. A functional interface can only define one abstract method, but it can define additional methods if they are default methods.) If `pred` is a predicate, then `pred.negate()` is a predicate that is true when pred is false and is false when pred is true. This means that `retain()` could actually be defined simply as

```
public static <T> void retain(Collection<T> coll,
Predicate<T> pred) {
    remove( coll, pred.negate() );
}
```

My test program creates some collections and applies the methods from the *Predicates* class to them. My collections contain objects of type *Integer* and I made two *Predicates* that work on objects of this type. One way to create such a predicate is to write a class that implements the interface *Predicate<Integer>*, and then to make an object belonging to that class. For example, I could have defined the class:

```
/**
 * An object of type Even tests an Object of type Integer
to see whether the
 * integer that it represents is an even number. Note that
the test() method
 * should only be applied to non-null values of type
Integer.
 */
static class Even implements Predicate<Integer> {
   public boolean test(Integer i) {
      if ( i % 2 == 0 )
         return true;
      else
         return false;
   }
} // end class Even
```

The predicate object could then created with a command:

```
Predicate<Integer> pred = new Even():
```

But of course, it is also possible to define the predicate more simply as a lambda expression:

```
Predicate<Integer> pred = i -> (i % 2 == 0);
```

or, of course, we could simply use the lambda expression at the point where the predicate is needed. For example:

```
result = Predicates.collect(coll, i -> (i % 2 == 0) )
```

I hope that you can see how using the lambda expression simplifies the code quite a bit!

The Solution

The Predicates Class:

```
import java.util.Collection;
import java.util.Iterator;
import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

/**
 * This class defines some static methods for working
 * with Collections and Predicates.  (The Predicate
 * interface is NOT a standard part of Java.)
 */
public class Predicates {


   /**
    * Remove every object, obj, from coll for which
pred.test(obj) is true.
    */
   public static <T> void remove(Collection<T> coll,
Predicate<T> pred) {
      Iterator<T> iter = coll.iterator();
      while (iter.hasNext()) {
         T item = iter.next();
         if (pred.test(item))
            iter.remove();
      }
   } // end remove()


   /**
    * 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> void retain(Collection<T> coll,
Predicate<T> pred){
      Iterator<T> iter = coll.iterator();
      while (iter.hasNext()) {
         T item = iter.next();
         if ( ! pred.test(item) )
            iter.remove();
      }
   } // end retain()
```

```
   /**
    * Return a List that contains all the objects, obj,
from the collection,
    * coll, for which pred.test(obj) is true.
    */
   public static <T> List<T> collect(Collection<T> coll,
Predicate<T> pred) {
      List<T> list = new ArrayList<T>();
      for ( T item : coll ) {
         if (pred.test(item))
            list.add(item);
      }
      return list;
   } // end collect()


   /**
    * 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.
    */
   public static <T> int find(ArrayList<T> list,
Predicate<T> pred) {
      //
      for (int i = 0; i < list.size(); i++) {
         T item = list.get(i);
         if (pred.test(item))
            return i;
      }
      return -1;
   } // end find()


} // end class Predicates
```

The Test program:

```
import java.util.*;
import java.util.function.Predicate;

/**
 * Perform some simple tests on the Predicates class.
 */
public class TestPredicates {
```

```
   /**
    * For convenience make a Collection containing some
integers.  The Collection
    * is actually a TreeSet, but that is not relevant to
the main program.
    */
   static Collection<Integer> makeSet() {
      Collection<Integer> set = new TreeSet<Integer>();
      set.add(Integer.valueOf(32));
      set.add(Integer.valueOf(17));
      set.add(Integer.valueOf(142));
      set.add(Integer.valueOf(56));
      set.add(Integer.valueOf(1899));
      set.add(Integer.valueOf(57));
      set.add(Integer.valueOf(999));
      set.add(Integer.valueOf(86));
      set.add(Integer.valueOf(83));
      set.add(Integer.valueOf(100));
      return set;
   } // end makeSet()

   /**
    * Main routine tests the Predicates class by making
Collections of Integers
    * and applying the methods from the Predicates class to
them.
    */
   public static void main(String[] args) {

      Collection<Integer> coll;    // A collection (from
makeSet() method).

      List<Integer> result;        // The result of
applying collect() to coll.

      Predicate<Integer> pred = i -> (i % 2 == 0);  //
Tests if an Integer is even.

      coll = makeSet();
      System.out.println("Original collection: " + coll);

      Predicates.remove(coll,pred);
      System.out.println("Remove evens: " + coll);

      coll = makeSet();
      Predicates.retain(coll,pred);
      System.out.println("Retain evens: " + coll);
```

```
      coll = makeSet();
      result = Predicates.collect(coll,pred);
      System.out.println("Collect evens: " + result);

      ArrayList<Integer> list = new ArrayList<Integer>();
      int i = Predicates.find(list,pred);
      System.out.println("Find first even: at index " + i);


      pred = n -> (n > 100);        // Tests if an Integer
is bigger than 100.

      coll = makeSet();
      System.out.println("Original collection: " + coll);

      Predicates.remove(coll,pred);
      System.out.println("Remove big: " + coll);

      coll = makeSet();
      Predicates.retain(coll,pred);
      System.out.println("Retain big: " + coll);

      coll = makeSet();
      result = Predicates.collect(coll,pred);
      System.out.println("Collect big: " + result);

      list = new ArrayList<Integer>(coll);
      i = Predicates.find(list,pred);
      System.out.println("Find first big: at index " + i);

   } // end main()

} // end class TestPredicates
```

Solution for Programming Exercise 10.5

Exercise 10.5:

This is a short exercise in using the stream API. Suppose that the class *Score* is defined as

```
/**
 * Data for one student about a score on a test.
 */
private record ScoreInfo(
    String lastName,
    String firstName,
    int score
) { };
```

defined here as a record class for convenience (see Section 7.4). And suppose that `scoreData` is an array of *ScoreInfos* containing information about the scores of students on a test. Use the stream API to do each of the following tasks:

- print the number of students (without using `scoreData.length`)
- print the average score for all of the students
- print the number of students who got an A (score greater than or equal to 90)
- use the `collect()` stream operation to create a *List<String>* that contains the names of students whose score was less than 70; the names should be in the form first name followed by last name
- print the names from the *List* that was generated in the previous task
- print out the students' names and scores, ordered by last name
- print out the students' names and scores, ordered by score

You can put all of the code in `main()` routine and include *ScoreInfo* as a nested class. Do not use **any** for loops or other control structures. Do everything using the stream API. For testing your code, you can use this array:

```
private static ScoreInfo[] scoreData = new ScoreInfo[] {
        new ScoreInfo("Smith","John",70),
        new ScoreInfo("Doe","Mary",85),
        new ScoreInfo("Page","Alice",82),
        new ScoreInfo("Cooper","Jill",97),
        new ScoreInfo("Flintstone","Fred",66),
        new ScoreInfo("Rubble","Barney",80),
        new ScoreInfo("Smith","Judy",48),
        new ScoreInfo("Dean","James",90),
        new ScoreInfo("Russ","Joe",55),
        new ScoreInfo("Wolfe","Bill",73),
        new ScoreInfo("Dart","Mary",54),
        new ScoreInfo("Rogers","Chris",78),
        new ScoreInfo("Toole","Pat",51),
        new ScoreInfo("Khan","Omar",93),
        new ScoreInfo("Smith","Ann",95)
};
```

Discussion

In each case, to process the values from the array using the stream API, we need to make a stream containing those values. That's done by calling `Arrays.stream(scoreData)`, except in the case of printing out the names in a *List*, where `list.stream()` can be used. From there, we just need to apply the appropriate stream operators to get the desired results. I have added some comments on each of the assigned tasks to the program.

The Solution

```
import java.util.List;
import java.util.Arrays;
import java.util.stream.Collectors;

/**
 * This program tests the stream API by doing various
computations.
 */
public class StreamTest {

    /**
     * Data for one student about a score on a test.
     */
```

```
    private record ScoreInfo(
        String lastName,
        String firstName,
        int score
    ) { };

    /**
     * A test array of StudentInfo that will be used to
create streams.
     */
    private static ScoreInfo[] scoreData = new ScoreInfo[]
{
            new ScoreInfo("Smith","John",70),
            new ScoreInfo("Doe","Mary",85),
            new ScoreInfo("Page","Alice",82),
            new ScoreInfo("Cooper","Jill",97),
            new ScoreInfo("Flintstone","Fred",66),
            new ScoreInfo("Rubble","Barney",80),
            new ScoreInfo("Smith","Judy",48),
            new ScoreInfo("Dean","James",90),
            new ScoreInfo("Russ","Joe",55),
            new ScoreInfo("Wolfe","Bill",73),
            new ScoreInfo("Dart","Mary",54),
            new ScoreInfo("Rogers","Chris",78),
            new ScoreInfo("Toole","Pat",51),
            new ScoreInfo("Khan","Omar",93),
            new ScoreInfo("Smith","Ann",95)
    };


    public static void main(String[] args) {

        /* The number of students can be computed simply by
applying
         * the .count() operation to a stream created from
the array.
         * This is more than a little silly, since the
answer will
         * be the same as scoreData.length, but it sets up
the
         * general format for the stream operations in the
rest of
         * the program. */

        long count =
Arrays.stream(scoreData).parallel().count();
        System.out.println("Number of students: " + count);
```

```
        /* To get the average, map the stream of ScoreData
objects
         * to an IntStream containing just the scores, and
then use
         * the .average() operation from the IntStream
class. Since
         * .average() returns an Optional<Double>, we need
to
         * use getAsDouble() to retrieve the final answer.
*/

        double avg = Arrays.stream(scoreData).parallel()
                                .mapToInt( s -> s.score )
                                .average()
                                .getAsDouble();
        System.out.printf("Average grade:  %1.2f%n", avg );

        /* To get the number of students who got an A,
first filter the
         * stream letting through only students who satisfy
the
         * predicate  s -> s.score >= 90.  This predicate
returns true
         * only for students whose score is greater than or
equal to 90.
         * The number is then computed by applying .count()
to the
         * resulting stream. */

        long countA = Arrays.stream(scoreData).parallel()
                   .filter( s -> s.score >= 90 )
                   .count();
        System.out.println("Number who got an A: " +
countA);
        System.out.println();

        /* To get a list of names of students who failed,
first filter
         * the stream to let through only ScoreInfo objects
with score
         * less than 70.  Since we want the names of
students, we need
         * to convert the stream of ScoreInfo objects into
a stream
         * of strings by applying a mapping operation.
Then applying
         * .collect(Collectors.toList()) produces the List
that we need. */
```

```
        List<String> failing = Arrays.stream(scoreData)
                                                .filter( s ->
(s.score < 70) )
                                                .map( s ->
(s.firstName + " " + s.lastName) )
                                                .collect(
Collectors.toList() );

        /* Now we need to print the list, which can easily
be done with
         * a for-each loop.  However, it can also be done
using .forEach()
         * operation on a stream.  In this case, the String
Consumer in the
         * .forEach() operation is given as the method
reference
         * System.out::println.  That is, it is the
println() method in the
         * object System.out.  This method reference is the
same as writing
         * the lambda expression  s ->
System.out.println(s).  */

        System.out.println("Failing students: ");
        failing.stream().forEach( System.out::println );
        System.out.println();

        /* To print the data ordered by last name, apply
the .sorted() operation
         * to sort the values in the stream by last name,
then use .forEach()
         * to output the information from each ScoreInfo
object in the stream.
         * The .sorted() operation takes a parameter that
is a Comparator for
         * comparing two values from the stream.  In this
case, we just
         * compare the lastNames from two ScoreInfo
objects, using the standard
         * compareTo() method for Strings. */

        System.out.println("Data sorted by last name:");
        Arrays.stream(scoreData)
             .sorted( (s1,s2) ->
s1.lastName.compareTo(s2.lastName) )
             .forEach( s -> System.out.printf(
                                  "  %s, %s: %d%n",
```

```
s.lastName, s.firstName, s.score) );
        System.out.println();

        /* The data can be output ordered by score in the
same way, using a
         * different comparator in the .sorted() operation.
*/

        System.out.println("Data sorted by score:");
        Arrays.stream(scoreData)
             .sorted( (s1,s2) -> s1.score - s2.score )
             .forEach( s -> System.out.printf(
                                 "  %s, %s: %d%n",
s.lastName, s.firstName, s.score) );

    }

} // end StreamTest
```

Solution for Programming Exercise 10.6

Exercise 10.6:

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.

Discussion

Solving this exercise, for the most part, means collecting subroutines that were presented in Subsection 10.4.2 into a complete program. I copied the method `addReference()` directly, just making it `static` and changing the names of the parameters to names that are more appropriate for a concordance. I copied `printIntegers()`, changed its name to `printConcordance`, made it static, and changed "System.out.print" to "TextIO.put" so that it would send its output to the file selected by the user.

In Subsection 10.4.3, I discussed the problem that arises when comparing strings that differ only in their use of upper and lower case. In that subsection, I used a *Comparator* to solve the problem. In my solution to this exercise, however, I used the simpler solution of converting all words to lower case immediately after reading them from the file. For lower case words, the default ordering is alphabetical order, so I don't need a *Comparator*.

It remains to write the `main()` routine, which is similar in outline to the `main()` routine in `WordCount.java`. One difference is that we have to keep track of line numbers. We can do this by looking for new-line characters as we skip over the characters between words. While we do this, we have to be on the alert for the special character `TextIO.EOF` that is returned by `TextIO.peek()` when the end of the file is reached. A pseudocode algorithm can be given as:

```
Open the input files.

Let lineNum = 1.

Repeat:

      // Skip over any non-letters in the input, stopping
when a
      // letter (marking the beginning of a word) or EOF
is found.
```

```
      while the next character is not end-of-file or a
letter:
         Read the next character.
         If it is a new line character:
            Count the line by adding 1 to lineNum

      // After the while loop, we are looking at either
the end of file
      //    or at a letter that is the beginning of the
next word.

      If at end-of-file:
         Exit from the loop.

      Get the next word from the input file.
      Convert the word to lower case.
      if the word is not "the" and has length > 2:
         Add the word and lineNum to the concordance.

Open the output file.

Print the concordance.
```

This can be translated directly (using some code from WordCount.java) to give the `main()` routine. For reading individual words from input, the method `readNextWord()` can be copied directly from WordCount.java. (In my program, I used a slightly simpler method named `readWord()` which uses the fact that when `readWord()` is called by the `main()` routine, the next character in input is already known to be a letter.)

The Solution

```
import textio.TextIO;

import java.util.Map;
import java.util.TreeMap;
import java.util.TreeSet;

/**
 * This program makes a concordance for a file.  A
concordance
 * lists all the words that occur in the file, along with
all
```

```
 * the line numbers on which each word occurs.  (Words of
 * length less than 3 are omitted, and "the" is omitted.)
The
 * concordance is written to an output file.  The user
selects
 * the input and output files using file dialog boxes.
This
 * program uses the non-standard class, TextIO.
 */
public class Concordance {



/**
 * This TreeMap holds the concordance.  Words from the file
 * are used as keys in the map.  The value associated with
 * each word is a set that contains the line numbers on
which
 * the word occurs in the file.  The set contains values
 * belonging to the wrapper class, Integer.
 */
private static TreeMap<String, TreeSet<Integer>>
concordance;



public static void main(String[] args) {

   System.out.println("""
                This program will ask you to select an
input file.
             It makes a list of all the words that occur in
the file
             along with the line number of each line that
contained
             that word.  This is called a \"concordance\"
for the file.
             After reading the input file, the program asks
you to
             select an output file.  If you select a file,
the list of
             words will be written to that file; if you
cancel, the list
             be written to standard output.  All words are
converted to
             lower case.

             Press return to begin.
```

```
            """);

   TextIO.getln();  // Wait for user to press return.

   try {

      // Let user select the input file.  If the user
cancels,
      // the program ends immediately.

      if (TextIO.readUserSelectedFile() == false) {
         System.out.println("No input file selected.
Exiting.");
         System.exit(0);
      }

      // Create the data structure that will hold the
concordance.

      concordance = new TreeMap<String, TreeSet<Integer>>
();

      int lineNum = 1;  // The number of the line in the
input
                        // file that is currently being
processed.

      // Read words from the file until end of file is
reached,
      // and add each word to the data.

      while (true) {
         char ch = TextIO.peek(); // Look ahead at next
character
         while ( ch != TextIO.EOF && !
Character.isLetter(ch) ) {
                   // Skip over non-letter characters,
stopping when
                   // end-of-file (TextIO.EOF) or a letter
is seen.  If the
                   // character is an end-of-line
character, add 1
                   // to lineNum to reflect the fact that
we are moving
                   // on to the next line in the file.
            TextIO.getAnyChar();  // Reads the next
character, which is junk.
            if (ch == '\n') {
```

```
               lineNum++;  // Start of a new line.
            }
            ch = TextIO.peek();  // Look at the next
character.
         }
         if (ch == TextIO.EOF) {
                 // The end-of-file has been reached, so
exit from the loop.
            break;
         }
         String word = readWord();  // The next word from
the file.
         word = word.toLowerCase();
         if (word.length() > 2  &&
!word.equalsIgnoreCase("the")) {
                 // Add the reference to word to the
concordance, unless
                 // the word is "the" or the word has
length <= 2.
            addReference(word,lineNum);
         }
      }

      // Write the data to a user-selected file, or to
standard
      // output if the user does not select an output file.

      System.out.println(concordance.size() + " distinct
words were found in the file.\n");
      System.out.println();
      if (concordance.size() == 0) {
         System.out.println("No words found in file.");
         System.out.println("Exiting without saving
data.");
         System.exit(0);
      }

      TextIO.writeUserSelectedFile(); // If user cancels,
output automatically
                                      // goes to standard
output.

      printConcordance();  // Print the data to the output
file.

   }
   catch (IllegalArgumentException e) {
      System.out.println( "Sorry, some error occurred:  " +
```

```
e.getMessage() );
   }

} // end main()


/**
 * Writes the data in the concordance to TextIO.  (The
output will go
 * to the output file, if one has been selected; otherwise,
it will go
 * to standard output.)  Each line of output contains one
word from the
 * file and a list of lines on which that word occurred.
The words
 * are in alphabetical order.
 */
private static void printConcordance() {

   for ( Map.Entry<String, TreeSet<Integer>>  entry :
concordance.entrySet() ) {

      String term = entry.getKey();
      TreeSet<Integer> pageSet = entry.getValue();

      TextIO.put( term + " " );
      for ( int page : pageSet ) {
         TextIO.put( page + " " );
      }
      TextIO.putln();

    }
}


/**
 * Add a word reference to the concordance.
 */
private static void addReference(String word, int lineNum)
{
   TreeSet<Integer> references; // The set of lines where
we have
                                //    previously found the
word.
   references = concordance.get(word);
   if (references == null){
          // This is the first reference that we have
          // found for the word.  Make a new set containing
```

```
          // the line number and add it to the concordance,
with
          // the word as the key.
       TreeSet<Integer> firstRef = new TreeSet<Integer>();
       firstRef.add( lineNum );  // lineNum is "autoboxed"
to give an Integer!
       concordance.put(word,firstRef);
   }
   else {
         // The variable references is the set of line
references
         // that we have found previously for the word.
         // Add the new line number to that set.  This
         // set is already associated to word in the
concordance.
      references.add( lineNum ); // pageNum is "autoboxed"
to give an Integer!
   }
}


/**
 * Read the next word from TextIO.  It is assumed that the
next character
 * in input is a letter.
 *    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 readWord() {
   char ch = TextIO.peek(); // Look at next character in
input.
   assert Character.isLetter(ch);
   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.
}


} // end class Concordance
```

Solution for Programming Exercise 10.7

Exercise 10.7:

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) {
           // (This uses the "switch expression" syntax)
       return switch(functionCode) {
               case SIN -> Math.sin(x);
               case COS -> Math.cos(x);
               case TAN -> Math.tan(x);
               case ABS -> Math.abs(x);
               case SQRT -> Math.sqrt(x);
               default -> 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(Function.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)`.

Discussion

The changes that must be made to SimpleInterpreter.java are remarkably small, aside from defining the *StandardFunction* class. In my solution, I included this class as a static nested class in my main program class. It would also make sense to define it as a separate class.

I changed the type of the map to *HashMap<String,Object>*, as suggested in the exercise. I added a few lines of code at the beginning of the `main()` routine to add the six standard functions to the symbol table, using statements similar to the example given in the exercise:

```
symbolTable.put("sin", new
StandardFunction(Functions.SIN));
symbolTable.put("cos", new
StandardFunction(Functions.COS));
symbolTable.put("tan", new
StandardFunction(Functions.TAN));
symbolTable.put("abs", new
StandardFunction(Functions.ABS));
symbolTable.put("sqrt", new
StandardFunction(Functions.SQRT));
symbolTable.put("log", new
StandardFunction(Functions.LOG));
```

The only other change is in the parsing routines. The change is in the `primaryValue()` method. When a word is encountered, the computer must check whether the word is a variable or a standard function. This is done, as suggested in the exercise, by looking up the word in the symbol table and checking the type of the associated value. If the word is a standard function, then we have to read and evaluate the argument of the function and then find the value of the function at that argument value:

```
else if ( Character.isLetter(ch) ) {
      // The factor is a variable or a standard function.
Read its name and
      // look up its value in the symbol table.  If the
```

```
name is not in the symbol table,
      // an error occurs.  (Note that the values in the
symbol table are objects of type
      // Double or StandardFunction.)
   String name = readWord();
   Object obj = symbolTable.get(name);
   if (obj == null)
      throw new ParseError("Unknown word \"" + name +
"\"");
   assert (obj instanceof Double || obj instanceof
StandardFunction);
   if (obj instanceof Double) {
         // The name is a variable; return value of that
variable.
      Double val = (Double)obj;
      return val.doubleValue();
   }
   else {
         // The name is a standard function.  Read the
argument
         // of the function and return the value of the
function
         // at that argument.  The argument must be an
expression
         // in parentheses.
      StandardFunction func = (StandardFunction)obj;
      TextIO.skipBlanks();
      if ( TextIO.peek() != '(' )
         throw new ParseError("Parenthesis missing after
standard function");
      TextIO.getChar(); // discard the '('
      double argument = expressionValue();  // read and
evaluate expression
      TextIO.skipBlanks();
      if ( TextIO.peek() != ')' )
         throw new ParseError("Missing right
parenthesis.");
      TextIO.getChar(); // discard the ')'
      return func.evaluate(argument);
   }
}
```

There is one additional change that I might have made. As the program stands, it allows the user to use a "let" command to assign a value to one of the standard function names. For example: "let sin = 42". This effectively

changes the name so that it is a variable, and the standard function is no longer available for use in expressions. It would probably be better to give an error message when the user tries to assign a value to a standard function name. This could be done in the `doLetCommand()` method. After reading the variable name in the "let" command, we have to check whether the name is already in the symbol table as the name of a standard function:

```
private static void doLetCommand() throws ParseError {
   TextIO.skipBlanks();
   if ( ! Character.isLetter(TextIO.peek()) )
      throw new ParseError("Expected variable name after
'let'.");
   String varName = readWord();  // The name of the
variable.
   Object symbol = symbolTable.get(name);
   if (symbol != null && symbol instanceof
StandardFunction)
       throw new ParseError("Can't assign a value to a
standard function name.");
   TextIO.skipBlanks();
   if ( TextIO.peek() != '=' )
      throw new ParseError("Expected '=' operator for
'let' command.");
   TextIO.getChar();
   double val = expressionValue();  // The value of the
variable.
   TextIO.skipBlanks();
   if ( TextIO.peek() != '\n' )
      throw new ParseError("Extra data after end of
expression.");
   symbolTable.put( varName, val );  // Add to symbol
table.
   System.out.println("ok");
}
```

The Solution

Changes from the program SimpleInterpreter.java are shown in red italic.

```
import textio.TextIO;

/*
```

This program can evaluate expressions that can include numbers, variables, parentheses, and the operators +, -, *, /, and ^ (where ^ indicates raising to a power), *and standard functions sin, cos, tan, abs, sqrt, and log.*

A variable name must consist of letters and digits, beginning with a letter. Names are case-sensitive.

This program accepts commands of two types from the user.

For a command of the form print <expression> , the expression is evaluated and the value is output. For a command of the form let <variable> = <expression> , the expression is evaluated and the value is assigned to the variable.

If a variable is used in an expression before it has been assigned a value, an error occurs. A number must begin with a digit (i.e., not a decimal point).

Commands are formally defined by the BNF rules:

```
<command>  ::=  "print" <expression>
                   |  "let" <variable> "="
<expression>

<expression>  ::=  [ "-" ] <term> [ [ "+" | "-"
] <term> ]...

<term>  ::=  <factor> [ [ "*" | "/" ] <factor>
]...

<factor>  ::=  <primary> [ "^" <factor> ]...

<primary>  ::=  <number> | <variable> | "("
<expression> ")"
                                | <standard-function-name>
"(" <expression> ")"
```

A line of input must contain exactly one such command. If extra data is found on a line after an expression has been read, it is considered an error. The variables "pi" and "e" are defined when the program starts to represent the usual

```
mathematical
    constants.

    This program demonstrates the use of a HashMap as a

symbol
    table.

    SimpleInterpreter2.java is based on the program
SimpleInterpreter.java,
    which did not handle standard functions in expressions.

    This program depends on the non-standard class, TextIO.
 */

import java.util.HashMap;

public class SimpleInterpreter2 {

   /**
    * 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


   /**
    * An enumerated type whose values represent the
possible
    * standard functions.
    */
   private enum Functions { SIN, COS, TAN, ABS, SQRT, LOG }


   /**
    * An object of this class represents one of the
standard functions.
    * Objects of this type are stored in the symbol table,
associated
    * with the name of the standard functions.  Note that
an object
    * of this type also knows how to evaluate the
corresponding function.
    */
   private static class StandardFunction {
```

```
   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) {
             // (This uses the switch expression syntax
from Subsection 3.6.5)
         return switch(functionCode) {
                 case SIN -> Math.sin(x);
                 case COS -> Math.cos(x);
                 case TAN -> Math.tan(x);
                 case ABS -> Math.abs(x);
                 case SQRT -> Math.sqrt(x);
                 default -> Math.log(x);
         };
      }

   } // end class StandardFunction



   /**
    * The symbolTable contains information about the values
of variables.  When a variable
    * is assigned a value, it is recorded in the symbol
table. The key is the name of the
    * variable, and the  value is an object of type Double
that contains the value of the
    * variable.
    *    The symbol table can also contain standard
```

```
functions.  The key is the name of
    * of the function, and the value is the corresponding
StandardFunction object.
    */
   private static HashMap<String,Object> symbolTable;


   public static void main(String[] args) {

      // Create the map that represents symbol table.

      symbolTable = new HashMap<String,Object>();

      // To start, add variables named "pi" and "e" to the
symbol
      // table.  Their values are the usual mathematical
constants.

      symbolTable.put("pi", Math.PI);
      symbolTable.put("e", Math.E);

      // Add the standard functions to the hash table.

      symbolTable.put("sin", new
StandardFunction(Functions.SIN));
      symbolTable.put("cos", new
StandardFunction(Functions.COS));
      symbolTable.put("tan", new
StandardFunction(Functions.TAN));
      symbolTable.put("abs", new
StandardFunction(Functions.ABS));
      symbolTable.put("sqrt", new
StandardFunction(Functions.SQRT));
      symbolTable.put("log", new
StandardFunction(Functions.LOG));

      System.out.println("\n\nEnter commands; press return
to end.");
      System.out.println("Commands must have the form:\n");
      System.out.println("      print <expression>");
      System.out.println("  or");
      System.out.println("      let <variable> =
<expression>");

      while (true) {
         System.out.print("\n?  ");
         TextIO.skipBlanks();
```

```
         TextIO.skipBlanks();
         if ( TextIO.peek() == '\n' ) {
            break;  // A blank input line ends the while
loop and the program.
         }
         try {
            String command = TextIO.getWord();
            if (command.equalsIgnoreCase("print"))
               doPrintCommand();
            else if (command.equalsIgnoreCase("let"))
               doLetCommand();
            else
               throw new ParseError("Command must begin
with 'print' or 'let'.");
            TextIO.getln();
         }
         catch (ParseError e) {
            System.out.println("\n*** Error in input:    "
+ e.getMessage());
            System.out.println("*** Discarding input:  " +
TextIO.getln());
         }
      }

      System.out.println("\n\nDone.");

   } // end main()


   /**
    * Process a command of the form  let <variable> =
<expression>.
    * When this method is called, the word "let" has
already
    * been read.  Read the variable name and the
expression, and
    * store the value of the variable in the symbol table.
    */
   private static void doLetCommand() throws ParseError {
      TextIO.skipBlanks();
      if ( ! Character.isLetter(TextIO.peek()) )
         throw new ParseError("Expected variable name after
'let'.");
      String varName = readWord();  // The name of the
variable.
      TextIO.skipBlanks();
      if ( TextIO.peek() != '=' )
         throw new ParseError("Expected '=' operator for
'let' command.");
```

```
 let command. ");
      TextIO.getChar();
      double val = expressionValue();  // The value of the
variable.
      TextIO.skipBlanks();
      if ( TextIO.peek() != '\n' )
         throw new ParseError("Extra data after end of
expression.");
      symbolTable.put( varName, val );  // Add to symbol
table.
      System.out.println("ok");
   }


   /**
    * Process a command of the form  print <expression>.
    * When this method is called, the word "print" has
already
    * been read.  Evaluate the expression and print the
value.
    */
   private static void doPrintCommand() throws ParseError {
      double val = expressionValue();
      TextIO.skipBlanks();
      if ( TextIO.peek() != '\n' )
         throw new ParseError("Extra data after end of
expression.");
      System.out.println("Value is " + val);
   }


   /**
    * Read an expression from the current line of input and
return its value.
    */
   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();
         negative = true;
      }
      double val;  // Value of the expression.
      val = termValue();  // An expression must start with
a term.
      if (negative)
```

```
      if (negative)
         val = -val; // Apply the leading minus sign
      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();
         double nextVal = termValue();
         if (op == '+')
            val += nextVal;
         else
            val -= nextVal;
         TextIO.skipBlanks();
      }
      return val;
   } // end expressionValue()


   /**
    * Read a term from the current line of input and return
its value.
    */
   private static double termValue() throws ParseError {
      TextIO.skipBlanks();
      double val;  // The value of the term.
      val = factorValue();  // A term must start with a
factor.
      TextIO.skipBlanks();
      while ( TextIO.peek() == '*' || TextIO.peek() == '/'
) {
            // Read the next factor, and multiply or divide
            // the value-so-far by the value of this
factor.
         char op = TextIO.getAnyChar();
         double nextVal = factorValue();
         if (op == '*')
            val *= nextVal;
         else
            val /= nextVal;
         TextIO.skipBlanks();
      }
      return val;
   } // end termValue()


   /**
```

```
   /
    * Read a factor from the current line of input and
return its value.
    */
   private static double factorValue() throws ParseError {
      TextIO.skipBlanks();
      double val;  // Value of the factor.
      val = primaryValue();  // A factor must start with a
primary.
      TextIO.skipBlanks();
      if ( TextIO.peek() == '^' ) {
            // Read the next factor, and exponentiate
            // the value by the value of that factor.
         TextIO.getChar();
         double nextVal = factorValue();
         val = Math.pow(val,nextVal);
         if (Double.isNaN(val))
            throw new ParseError("Illegal values for ^
operator.");
         TextIO.skipBlanks();
      }
      return val;
   } // end termValue()


   /**
    *  Read a primary from the current line of input and
    *  return its value.  A primary must be a number,
    *  a variable, or an expression enclosed in
parentheses.
    */
   private static double primaryValue() throws ParseError {
      TextIO.skipBlanks();
      char ch = TextIO.peek();
      if ( Character.isDigit(ch) ) {
            // The factor is a number.  Read it and
            // return its value.
         return TextIO.getDouble();
      }
      else if ( Character.isLetter(ch) ) {
            // The factor is a variable or a standard
function.  Read its name and
            // look up its value in the symbol table.  If
the name is not in the symbol table,
            // an error occurs.  (Note that the values in
the symbol table are objects of type
            // Double or StandardFunction.)
         String name = readWord();
```

```
         Object obj = symbolTable.get(name);
         if (obj == null)
            throw new ParseError("Unknown word \"" + name +
"\"");

         assert (obj instanceof Double || obj instanceof
StandardFunction);
         if (obj instanceof Double) {
               // The name is a variable; return value of
that variable.
            Double val = (Double)obj;
            return val.doubleValue();
         }
         else {
               // The name is a standard function.  Read
the argument
               // of the function and return the value of
the function
               // at that argument.  The argument must be
an expression
               // in parentheses.
            StandardFunction func = (StandardFunction)obj;
            TextIO.skipBlanks();
            if ( TextIO.peek() != '(' )
               throw new ParseError("Parenthesis missing
after standard function");
            TextIO.getChar(); // discard the '('
            double argument = expressionValue();  // read
and evaluate expression
            TextIO.skipBlanks();
            if ( TextIO.peek() != ')' )
               throw new ParseError("Missing right
parenthesis.");
            TextIO.getChar(); // discard the ')'
            return func.evaluate(argument);
         }
      }
      else if ( ch == '(' ) {
            // The factor is an expression in parentheses.
            // Return the value of the expression.
         TextIO.getAnyChar();  // Read the "("
         double val = expressionValue();
         TextIO.skipBlanks();
         if ( TextIO.peek() != ')' )
            throw new ParseError("Missing right
parenthesis.");
         TextIO.getAnyChar();  // Read the ")"
```

```
         return val;
      }
      else if ( ch == '\n' )
         throw new ParseError("End-of-line encountered in
the middle of an expression.");
      else if ( ch == ')' )
         throw new ParseError("Extra right parenthesis.");
      else if ( ch == '+' || ch == '-' || ch == '*' || ch
== '/')
         throw new ParseError("Misplaced operator.");
      else
         throw new ParseError("Unexpected character \"" +
ch + "\" encountered.");
   }


   /**
    *  Reads a word from input.  A word is any sequence of
    *  letters and digits, starting with a letter.  When
    *  this subroutine is called, it should already be
    *  known that the next character in the input is
    *  a letter.
    */
   private static String readWord() {
      String word = "";  // The word.
      char ch = TextIO.peek();
      while (Character.isLetter(ch) ||
Character.isDigit(ch)) {
         word += TextIO.getChar(); // Add the character to
the word.
         ch = TextIO.peek();
      }
      return word;
   }

} // end class SimpleInterpreter2
```

Solution for Programming Exercise 11.1

Exercise 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.
```

Discussion

One possible recursive algorithm for printing all the files in a directory and its subdirectories is:

```
Print the name of the directory
for each file in the directory:
   if the file is a directory:
      Print its contents recursively
   else
      Print the name of the file.
```

If the directory is given as an object of type *File*, then this can be coded easily. We need some instance methods from the *File* class. (This subroutine is not quite what I ended up using in my solution.)

```
static void listDirectoryContents( File dir ) {
    System.out.println("Directory \"" + dir.getName() +
```

```
"\"");
    String[] files;  // The names of the files in the
directory.
    files = dir.list();
    for (int i = 0; i < files.length; i++) {
        File f;  // One of the files in the directory.
        f = new File(dir, files[i]);
        if ( f.isDirectory() ) {
               // Call listDirectoryContents() recursively
to
               // list the contents of the directory, f.
            listDirectoryContents(f);
        }
        else {
              // For a regular file, just print the name,
files[i].
            System.out.println(files[i]);
        }
    }
} // end listDirectoryContents()
```

Every time this routine finds a directory, it lists not just the name of the directory but also, recursively, the names of everything that it contains. The only problem with this is that it doesn't indicate which items are in which directory. For example, consider the output:

```
Directory "games"
blackbox
Directory "CardGames"
cribbage
euchre
tetris
```

There is no way to tell where the list of items in "CardGames" ends. Possibly, for example, "euchre" is in the "CardGames" directory. But possibly, "cribbage" is the only file in "CardGames" and "euchre" is actually part of the listing for "games". It would be nice to use indentation to show the nesting, like this:

```
Directory "games"
   blackbox
   Directory "CardGames"
      cribbage
      euchre
   tetris
```

In this listing, you can tell that "euchre" is in fact in "CardGames" while "tetris" is in "games". To implement this, we just have to recognize that the indentation can be different in each call to the `listDirectoryContents()` method. Since it can be different, it should be a parameter. When we call `listDirectoryContents()` recursively, we should increase the indentation. You can see how this is done in the actual solution, given below.

The Solution

```
import java.io.File;
import java.util.Scanner;

/**
 * This program lists the files in a directory specified by
 * the user.  If one of those files is itself a directory,
the
 * program recursively lists the contents of that
subdirectory.
 * The user is asked to type in the name of the directory
that is
 * to be listed.  If the name entered by the user is not a
directory,
 * a message is printed and the program ends.
 */
public class RecursiveDirectoryList {


   public static void main(String[] args) {

      String directoryName;  // Directory name entered by
the user.
      File directory;        // File object referring to
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 // the file exists but is not a directory
            System.out.println("That file is not a
directory.");
      }
      else {
            // List the contents of the directory
recursively, with
            // no indentation at the top level.
         listDirectoryContents(directory,"");
      }

   } // end main()


   /**
    * A recursive subroutine that lists the contents of a
directory,
    * including the contents of its subdirectories to any
level of nesting.
    * @param dir the directory whose contents are to be
listed.  It is assumed
    *   that dir is in fact a directory.
    * @param indent a string of blanks that is prepended to
each item in
    *    the directory listing.  This string grows in
length with each increase
    *    in the level of directory nesting.
    */
   private static void listDirectoryContents(File dir,
String indent) {
      String[] files;  // List of names of files in the
directory.
      System.out.println(indent + "Directory \"" +
dir.getName() + "\":");
      indent += "   ";  // Increase the indentation for
listing the contents.
      files = dir.list();
      for (int i = 0; i < files.length; i++) {
            // If the file is a  directory, list its
contents
            // recursively.  Otherwise, just print its
name.
         File f = new File(dir, files[i]);
         if (f.isDirectory())
            listDirectoryContents(f, indent);
```

```
         else
            System.out.println(indent + files[i]);
      }
   } // end listContents()


} // end class RecursiveDirectoryList
```

Solution for Programming Exercise 11.2

Exercise 11.2:

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* or a *BufferedReader* to process each file.

Discussion

The `main()` routine for this program is a simple for loop that processes the command-line arguments. Each argument is supposed to be the name of a file. The loop just prints the file name and calls a subroutine that counts the number of lines in the file and outputs the result:

```
for (int i = 0; i < args.length; i++) {
   System.out.print(args[i] + ": ");
   countLines(args[i]);
}
```

The `countLines()` subroutine will catch any errors that occur when it tries to access the file. If an error occurs, it will output an error message instead of the number of lines. Since the error is handled in the subroutine, it won't crash the program or stop the `main()` routine from going on to process any remaining files.

The `countLines()` subroutine can be written using either a a *BufferedReader* or a *Scanner* to read lines from the file; each of these classes has a simple method for reading a complete line of text from the file. (An alternative approach would read individual characters from the file and count the number of end-of-line markers. Unfortunately, ends-of-line can be marked by any of the strings "\n", "\r", or "\r\n" and that would add to the difficulty of the task. The line reading methods in *BufferedReader* and *Scanner* will work no matter which end-of-line marker is used.)

In my solution, I use a *BufferedReader*. The `countLines()` method creates a *BufferedReader* stream to read from the file. It then calls the `readLine()` routine from the *BufferedReader* class to read lines of text from the file, until the end of the file is encountered. For a *BufferedReader*, the end-of-file can be recognized when `readLine()` returns `null`. The method counts each line as it reads it. In the end, the value of the counter is the number of lines in the file. The subroutine writes this value to standard output. All this is done in a `try..catch` statement, treating the *BufferedReader* as a resource, so that an error can be handled if it occurs. It's pretty straightforward. The complete solution is given below.

An interesting variation on this would be to use the stream API from Section 10.6. A *BufferedReader*, `in`, has a method `in.lines()` that creates a stream of strings containing the lines of the file. We can find out how many lines there are just by applying the `count()` stream operation to that stream. So, using the stream API, the lines could be counted using

```
long lineCt = in.lines().count();
```

One note about the command line for calling this program. If you are using UNIX, including MacOS or Linux, you can take advantage of something called "wildcards" in the command-line arguments. For example, if you say, "`java LineCounts *.txt`", the "*" is a wildcard. The operating system will expand "*.txt" into a list of all the files in the current directory that end in ".txt". Similarly, "*xx*" expands into a list of all file names that contain "xx", and "fil*dat" expands into the list of file names that begin with "fil" and end with "dat". The "*" matches any number of characters in the file name, including zero characters. This expansion is done before your program sees the command-line arguments, so you don't have to worry about it in your

program. Typing the command "`java LineCounts *.txt`" would be exactly equivalent to typing something like "`java LineCounts file1.txt file2.txt file3.txt`". This type of expansion happens for any command in UNIX, not just for the "java" command.

I should also note that if a file name contains spaces or other special characters, then the file name should be enclosed in quotation marks when it is used as an argument on the command line.

The Solution

```
import java.io.*;

/**
 * This program reports the number of lines in each of the
files
 * that are specified as command line arguments.  The files
are
 * assumed to be text files.  If a file does not exist, or
if
 * some error occurs when the attempt is made to read the
file,
 * then an error message is printed (but the other files
are
 * still processed).
*/
public class LineCounts {


   /**
    * The main() routine simply gets the file names from
the
    * command line and calls the countLines() routine to
process
    * each name.  Since any errors are handled in the
countLines()
    * routine, the main program will continue after an
error occurs
    * while trying to process one of the files.
    */
   public static void main(String[] args) {

      if (args.length == 0) {
```

```
            // This program must have at least one
command-line
            // argument to work with.
         System.out.println("Usage:   java LineCounts
<file-names>");
         return;
      }

      for (int i = 0; i < args.length; i++) {
         System.out.print(args[i] + ":  ");
         countLines(args[i]);
      }

   }  // end main()


   /**
    * Count the number of lines in the specified file, and
    * print the number to standard output.  If an error
occurs
    * while processing the file, print an error message
instead.
    * Two try..catch statements are used so I can give a
    * different error message in each case.
    */
   private static void countLines(String fileName) {

      try(BufferedReader in = new BufferedReader( new
FileReader(fileName) )) {

         int lineCount = 0;  // number of lines read from
the file

         String line = in.readLine();   // Read the first
line.
         while (line != null) {
             lineCount++;               // Count this line.
             line = in.readLine();      // Read the next
line.
         }
         System.out.println(lineCount + " lines");

      }
      catch (FileNotFoundException e) {
          System.out.println("Error.  Can't open file.");
      }
      catch (Exception e) {
         System.out.println("Error.   Problem with reading
```

```
from file.");
      }

   }  // end countLines()


} // end class LineCounts
```

Solution for Programming Exercise 11.3

Exercise 11.3:

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 a line beginning with the word "ERROR" to the client and closes the connection. (The error response can include an error message on the rest of the line.)

Your program should use a subroutine to handle each request that the server receives. It should not stop after handling one request; it should remain open and continue to accept new requests. See the DirectoryList example in Subsection 11.2.2 for help with the problem of getting the list of files in the directory.

Discussion

The sample program DateServer.java, from Subsection 11.4.4, shows the typical outline of a server program. For this exercise, the processing of each

connection request is more complicated, but the basic outline of the program is the same.

My program begins by getting the directory name from the command line. An object of type *File* is created which can be used to check whether the specified directory exists and is in fact a directory rather than a regular file. This allows for some basic error checking before the server starts running:

```
if (args.length == 0) {
   System.out.println("Usage:  java FileServer
<directory>");
   return;
}
directory = new File(args[0]);
if ( ! directory.exists() ) {
   System.out.println("Specified directory does not
exist.");
   return;
}
if (! directory.isDirectory() ) {
   System.out.println("The specified file is not a
directory.");
   return;
}
```

The *File* object is needed later, in any case, for reading the list of files in the directory.

The rest of the `main()` routine takes the typical form for a server program. A listening socket is established. Then the server accepts and processes connection requests in an infinite loop. In this case, the processing is done by calling a `handleConnection()` method that is defined later in the program. The actual communication with the client is done in this method. (Since it can potentially take a long time to transmit a file, it would be desirable to use threads to handle the connections in this program, instead of running them in the same thread that listens for connection requests. That would allow several requests to be handled simultaneously. Using threads for network communication will be covered in the next chapter.

The `handleConnection()` method implements the communication protocol of the server in a straightforward way: It reads a line from the client. If the

line is "index", it sends a list of available files. If the line starts with the string "get", it sends the requested file. Finally, it closes the connection. (In fact, the connection is closed in the `finally` clause of a `try` statement to make sure that the connection is closed before the method returns.) You should be able to follow all this in the solution that follows.

(One minor point: Suppose that the last line in a file is not terminated by an end-of-line marker. That is, there is no end-of-line directly preceding the end-of-file. When my program transmits the file, it will nevertheless send an end-of-line marker after the last line of the file. So, the file that is transmitted is not identical to the file on the computer's hard disk. Actually, things are even worse than that because the end-of-line markers used when the file is transmitted are not necessarily the same as the ones that are used in the file as stored on the disk. It would probably be better to consider the file to be binary data and to send an exact, byte-by-byte copy.)

The Solution

```
import java.net.*;
import java.io.*;
import java.util.Scanner;

/**
 * This program is a very simple network file server.  The
 * server has a list of available text files that can be
 * downloaded by the client.  The client can also download
 * the list of files.  When the connection is opened, the
 * client sends one of two possible commands to the server:
 * "INDEX" or "GET <file-name>".  The server replies to
 * the first command by sending the list of available
files.
 * It responds to the second with a one-line message,
 * either "OK" or "ERROR".  If the message is "OK", it is
 * followed by the contents of the file with the specified
 * name.  The "ERROR" message indicates that the specified
 * file does not exist on the server. (The server can also
 * respond with the message "ERROR unsupported command" if
the
 * command it reads is not one of the two possible legal
commands.)
 * (The commands INDEX and GET are not case-sensitive.)
```

```
 *
 * The server program requires a command-line parameter
 * that specifies the directory that contains the files
 * that the server can serve.  The files should all be
 * text files, but this is not checked.  Also, the server
 * must have permission to read all the files.
 */
public class FileServer {

   static final int LISTENING_PORT = 3210;


   public static void main(String[] args) {

      File directory;        // The directory from which
the server
                             //    gets the files that it
serves.

      ServerSocket listener; // Listens for connection
requests.

      Socket connection;     // A socket for communicating
with a client.


      /* Check that there is a command-line argument.
         If not, print a usage message and end. */

      if (args.length == 0) {
         System.out.println("Usage:  java FileServer
<directory>");
         return;
      }

      /* Get the directory name from the command line, and
make
         it into a file object.  Check that the file exists
and
         is in fact a directory. */

      directory = new File(args[0]);
      if ( ! directory.exists() ) {
         System.out.println("Specified directory does not
exist.");
         return;
      }
      if (! directory.isDirectory() ) {
```

```
         System.out.println("The specified file is not a
directory.");
         return;
      }

      /* Listen for connection requests from clients.  For
         each connection, call the handleConnection()
method
         to process it.  The server runs until the program
         is terminated, for example by a CONTROL-C. */

      try {
         listener = new ServerSocket(LISTENING_PORT);
         System.out.println("Listening on port " +
LISTENING_PORT);
         while (true) {
            connection = listener.accept();
            handleConnection(directory,connection);
         }
      }
      catch (Exception e) {
         System.out.println("Server shut down
unexpectedly.");
         System.out.println("Error:  " + e);
         return;
      }

   } // end main()


   /**
    * This method processes the connection with one client.
    * It creates streams for communicating with the client,
    * reads a command from the client, and carries out that
    * command.  The connection is also logged to standard
output.
    * An output beginning with ERROR indicates that a
network
    * error occurred.  A line beginning with OK means that
    * there was no network error, but does not imply that
the
    * command from the client was a legal command.
    */
   private static void handleConnection(File directory,
Socket connection) {
      Scanner incoming;       // For reading data from the
client.
      PrintWriter outgoing;   // For transmitting data to
```

```
the client.
      String command = "Command not read";
      try {
         incoming = new Scanner(
connection.getInputStream() );
         outgoing = new PrintWriter(
connection.getOutputStream() );
         command = incoming.nextLine();
         if (command.equalsIgnoreCase("index")) {
            sendIndex(directory, outgoing);
         }
         else if (command.toLowerCase().startsWith("get")){
            String fileName = command.substring(3).trim();
            sendFile(fileName, directory, outgoing);
         }
         else {
            outgoing.println("ERROR unsupported command");
            outgoing.flush();
         }
         System.out.println("OK    " +
connection.getInetAddress()
               + " " + command);
      }
      catch (Exception e) {
         System.out.println("ERROR " +
connection.getInetAddress()
               + " " + command + " " + e);
      }
      finally {
         try {
            connection.close();
         }
         catch (IOException e) {
         }
      }
   }

   /**
    * This is called by the handleConnection() method in
response to an "INDEX" command
    * from the client.  Send the list of files in the
server's directory.
    */
   private static void sendIndex(File directory,
PrintWriter outgoing) throws Exception {
      String[] fileList = directory.list();
      for (int i = 0; i < fileList.length; i++)
         outgoing.println(fileList[i]);
```

```
      outgoing.flush();
      outgoing.close();
      if (outgoing.checkError())
         throw new Exception("Error while transmitting
data.");
   }

   /**
    * This is called by the handleConnection() command in
response to "GET <fileName>"
    * command from the client.  If the file doesn't exist,
send the message "ERROR".
    * Otherwise, send the message "OK" followed by the
contents of the file.
    */
   private static void sendFile(String fileName, File
directory, PrintWriter outgoing)

throws Exception {
      File file = new File(directory,fileName);
      if ( (! file.exists()) || file.isDirectory() ) {
         // (Note:  Don't try to send a directory, which
         // shouldn't be there anyway.)
         outgoing.println("ERROR");
      }
      else {
         outgoing.println("OK");
         BufferedReader fileIn = new BufferedReader( new
FileReader(file) );
         while (true) {
            // Read and send lines from the file until
            // an end-of-file is encountered.
            String line = fileIn.readLine();
            if (line == null)
               break;
            outgoing.println(line);
         }
      }
      outgoing.flush();
      outgoing.close();
      if (outgoing.checkError())
         throw new Exception("Error while transmitting
data.");
   }


} //end class FileServer
```

Solution for Programming Exercise 11.4

Exercise 11.4:

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).

Discussion

One possible user interface would be to present the user with a menu of actions that the program can perform. For variety, however, I decided on an interface based entirely on the command line. The first command line argument must be the name or IP address of the computer where a copy of the *FileServer* program from Exercise 11.3 is running. If that is the only command-line argument, then the client will contact the server and send an "index" command to the server. The server responds with a list of file names. The client reads these names and displays them on the screen. The program then ends. (If you want to give another command, you have to run the client program again with a new command line.)

If there are two command-line arguments, then the second argument must be the name of a file on the server. The client contacts the server and sends a "GET" command. The server responds with a one-line message, either an "ERROR" or "OK". The client reads this message. If the message starts with "ERROR", indicating that the requested file can't be sent, then the client just displays an error message to the user. If the message is "OK", then the server also sends the contents of the file. The client tries to create a local file of the same name. It reads the data from the server and saves it to that file. However, for safety, the client will not create a new file if a local file of the

same name already exists. This is considered to be an error. (An alternative would be to ask the user whether to replace the existing file.)

Finally, I wanted to make it possible to save a downloaded file in a local file with a different name. For this, three command line arguments are used. The first is the server, the second is the name of the file on the server, and the third is the name of the local file where the downloaded file is to be saved. In this case, the program is willing to overwrite an existing file of the same name, so the command must be used with some care.

If the server program is running on the same computer as the client (for demonstration purposes), the following command lines can be used to run the client:

```
java FileClient  localhost
java FileClient  localhost  datafile.txt
java FileClient  localhost  datafile.txt  mycopy.dat
java FileClient  localhost  datafile.txt  datafile.txt
```

The first command shows a list of files that are available on the server. The other three all try to get a file named "datafile.txt" from the server. The second command would refuse the save the file, if a file named datafile.txt already exists. The third command would save the file with the name "mycopy.dat." The last command would replace any existing datafile.txt with the file retrieved from the server.

The actual programming of the client is fairly straightforward. The example program DateClient.java, from Subsection 11.4.4 provides a model for opening a connection to the server. You should be able to follow the solution, given below.

Note that the client program must know the protocol that is used to communicate with the server. The **user** of the program, however, doesn't need to know anything about the protocol. The user only has to know how to use the program, and the user interface does not reflect or depend on the details of the protocol.

By the way, once you understand how the *FileClient* and *FileServer* examples work, it's not a big conceptual leap to understanding how the World

Wide Web works. A Web server program is just a greatly souped-up version of the *FileServer* program. It has access to a collection of files. It receives and responds to requests for those files from client programs. To get a file, the client program—a Web browser—needs to know the computer on which the server is running and the name of the file. This information is encoded in the url address of a Web page—just like it's given on the command line of the *FileClient* program. A Web page, of course, can contain links to other Web pages. The link includes a url with the necessary information for finding the page. To get the page, the Web browser just goes through the same process of contacting the specified server and requesting the specified file. (One big complication is that not all the files on a Web server are text files, so the client needs some way of knowing what type of data is stored in the file, and it needs to know how to handle data of that type. Another complication is that Web servers support "server side programming." That is, in response to certain URLs, instead of just sending a file, the Web server will run a program and send the output to the client.)

The Solution

```
import java.net.*;
import java.io.*;

/**
 * This program is a client for the FileServer server.  The
 * server has a list of available text files that can be
 * downloaded by the client.  The client can also download
 * the list of files.  When the connection is opened, the
 * client sends one of two possible commands to the server:
 * "INDEX" or "GET <file-name>".  The server replies to
 * the first command by sending the list of available
files.
 * It responds to the second with a one-line message,
 * either "OK" or "ERROR".  If the message is "OK", it is
 * followed by the contents of the file with the specified
 * name.  The "ERROR" message indicates that the specified
 * file does not exist on the server.
 *
 * The client program works with command-line arguments.
 * The first argument must be the name or IP address of the
 * computer where the server is running.  If that is the
 * only argument on the command line, then the client
 * gets the list of files from the server and displays
```

```
 * gets the list of files from the server and displays
 * it on standard output.  If there are two parameters,
 * the second parameter is interpreted as the name of a
 * file to be downloaded.  A copy of the file is saved
 * as a local file of the same name, unless a file of
 * the same name already exists.  If there are three
 * arguments, the second is the name of the file to be
 * downloaded and the third is the name under which the
 * local copy of the file is to be saved.  This will
 * work even if a file of the same name already exists.
 */
public class FileClient {

   static final int LISTENING_PORT = 3210;


   public static void main(String[] args) {

      String computer;          // Name or IP address of
server.
      Socket connection;        // Socket for communicating
with that computer.
      PrintWriter outgoing;     // Stream for sending a
command to the server.
      BufferedReader incoming;  // Stream for reading data
from the connection.
      String command;           // Command to send to the
server.


      /* Check that the number of command-line arguments is
legal.
         If not, print a usage message and end. */

      if (args.length == 0 || args.length > 3) {
         System.out.println("Usage:  java FileClient
<server>");
         System.out.println("    or  java FileClient
<server> <file>");
         System.out.println(
               "    or  java FileClient <server> <file>
<local-file>");
         return;
      }

      /* Get the server name and the message to send to the
server. */
```

```
      computer = args[0];

      if (args.length == 1)
         command = "INDEX";
      else
         command = "GET " + args[1];

      /* Make the connection and open streams for
communication.
         Send the command to the server.  If something
fails
         during this process, print an error message and
end. */

      try {
         connection = new Socket( computer, LISTENING_PORT
);
         incoming = new BufferedReader(
                            new
InputStreamReader(connection.getInputStream()) );
         outgoing = new PrintWriter(
connection.getOutputStream() );
         outgoing.println(command);
         outgoing.flush(); // ESSENTIAL: Make sure command
is dispatched to server!
      }
      catch (Exception e) {
         System.out.println(
              "Can't make connection to server at \"" +
args[0] + "\".");
         System.out.println("Error:  " + e);
         return;
      }

      /* Read and process the server's response to the
command. */

      try {
         if (args.length == 1) {
               // The command was "index".  Read and
display lines
               // from the server until the end-of-stream
is reached.
            System.out.println("File list from server:");
            while (true) {
               String line = incoming.readLine();
               if (line == null)
                   break;
```

```
               System.out.println("   " + line);
            }
         }
         else {
               // The command was "get <file-name>".  Read
the server's
               // response message.  If the message is
"OK", get the file.
            String message = incoming.readLine();
            if (! message.equalsIgnoreCase("OK")) {
               System.out.println("File not found on
server.");
               System.out.println("Message from server: " +
message);
               return;
            }
            PrintWriter fileOut;  // For writing the
received data to a file.
            if (args.length == 3) {
                  // Use the third parameter as a file
name.
                  // This will overwrite a file with the
same name!
                fileOut = new PrintWriter( new
FileWriter(args[2]) );
            }
            else {
                  // Use the second parameter as a file
name,
                  // but don't replace an existing file.
                File file = new File(args[1]);
                if (file.exists()) {
                   System.out.println("A file with that
name already exists.");
                   System.out.println("To replace it, use
the three-argument");
                   System.out.println("version of the
command.");
                   return;
                }
                fileOut = new PrintWriter( new
FileWriter(args[1]) );
            }
            while (true) {
                   // Copy lines from incoming to the file
until
                   // the end of the incoming stream is
encountered.
```

```
                String line = incoming.readLine();
                if (line == null)
                    break;
                fileOut.println(line);
            }
            if (fileOut.checkError()) {
               System.out.println("Some error occurred
while writing the file.");
               System.out.println("Output file might be
empty or incomplete.");
            }
            else {
               System.out.println("File has been
received.");
            }
         }
      }
      catch (Exception e) {
         System.out.println(
                 "Sorry, an error occurred while reading
data from the server.");
         System.out.println("Error: " + e);
      }
      finally {
         try {
            connection.close();
         }
         catch (IOException e) {
         }
      }

   }  // end main()


} //end class FileClient
```

Solution for Programming Exercise 11.5

Exercise 11.5:

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 11.5.2. 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 a short exercise in basic XML processing; as before, the program in this exercise is not meant to be a useful phone directory program.)

Discussion

The data for the phone book is stored in a *TreeMap* in which the entries are name/number pairs. The original program uses a two-line for-each loop to output the data to a *PrintWriter* named `out`:

```
for ( Map.Entry<String,String> entry :
phoneBook.entrySet() )
   out.println(entry.getKey() + "%" + entry.getValue() );
```

The data file is written at the end of the program. Writing out the data in the XML format shown in the exercise is only a little harder. The same values are output, but we have to add in a lot of extra text:

```
out.println("<?xml version=\"1.0\"?>");
out.println("<phone_directory>");
for ( Map.Entry<String,String> entry :
phoneBook.entrySet() ) {
   out.print("  <entry name='");
   out.print(entry.getKey());
   out.print("' number='");
   out.print(entry.getValue());
   out.println("'/>");
}
out.println("</phone_directory>");
```

The data file is read at the beginning of the program. To use the Document Object Model for this, we have to create a *DocumentBuilder* and use it to parse the data file. The code comes directory from Subsection 11.5.2:

```
DocumentBuilder docReader =

DocumentBuilderFactory.newInstance().newDocumentBuilder();
Document xmldoc = docReader.parse(dataFile);
```

If this succeeds, we know that the file contained a well-formed XML document. We can then call `xmldoc.getDocumentElement()` to get the root element of the document. This element should have tag name "phone_directory." The phone directory entries are stored in elements that are directly nested inside the root element. We can get a list of the children of the root element by calling `root.getChildNodes()` and we can traverse that list to get the individual directory entries. The name and number for an entry are stored as values of attributes named "name" and "number", and we can retrieve them using the `getAttribute()` method. Once we have a name and number, we can add it to the *TreeMap* in the usual way. Here is the code that gets the data from the DOM representation of the XML data file:

```
Element root = xmldoc.getDocumentElement();
if (! root.getTagName().equals("phone_directory"))
   throw new Exception();
NodeList nodes = root.getChildNodes();
for (int i = 0; i < nodes.getLength(); i++) {
   if ( nodes.item(i) instanceof Element ) {
      Element entry = (Element)nodes.item(i);
      if (! entry.getTagName().equals("entry"))
         throw new Exception();
      String entryName = entry.getAttribute("name");
```

```
        String entryNumber = entry.getAttribute("number");
        if (entryName.length() == 0 || entryNumber.length()
== 0)
           throw new Exception();
        phoneBook.put(entryName,entryNumber);
    }
}
```

A complete solution is given below, with changes from PhoneDirectoryFileDemo.java shown in red italic.

The Solution

```
import java.io.*;

import java.util.Map;
import java.util.TreeMap;
import java.util.Scanner;

import javax.xml.parsers.*;
import org.w3c.dom.*;

/**
 * This program lets the user keep a persistent "phone
book" that
 * contains names and phone numbers.  The data for the
phone book
 * is stored in a file in the user's home directory.
 *
 * The program is meant only as a demonstration of files
and XML.
 * The data file uses XML to represent the phone book data.
 */
public class PhoneDirectoryXMLDemo {

   /**
    * The name of the file in which the phone book data is
kept.  The
    * file is stored in the user's home directory.  The "."
at the
    * beginning of the file name means that the file will
be a
    * "hidden" file on Unix-based computers, including
Linux and
    * MacOS.
```

```
    */
   private static String DATA_FILE_NAME =
".phone_book_xml_demo";


   public static void main(String[] args) {

      String name, number;  // Name and number of an entry
in the directory
                            // (used at various places in
the program).

      TreeMap<String,String>  phoneBook;   // Phone
directory data structure.
                                           // Entries are
name/number pairs.

      phoneBook = new TreeMap<String,String>();


      /* Create a dataFile variable of type File to
represent the
       * data file that is stored in the user's home
directory.
       */

      File userHomeDirectory = new File(
System.getProperty("user.home") );
      File dataFile = new File( userHomeDirectory,
DATA_FILE_NAME );


      /* If the data file already exists, then the data in
the file is
       * read and is used to initialize the phone
directory.
       */

      if ( ! dataFile.exists() ) {
         System.out.println("No phone book data file found.
A new one");
         System.out.println("will be created, if you add
any entries.");
         System.out.println("File name:  " +
dataFile.getAbsolutePath());
      }
      else {
```

```
         System.out.println("Reading phone book data...");
         try {
            DocumentBuilder docReader =

DocumentBuilderFactory.newInstance().newDocumentBuilder();
            Document xmldoc = docReader.parse(dataFile);
            Element root = xmldoc.getDocumentElement();
            if (!
root.getTagName().equals("phone_directory"))
               throw new Exception();
            NodeList nodes = root.getChildNodes();
            for (int i = 0; i < nodes.getLength(); i++) {
               if ( nodes.item(i) instanceof Element ) {
                  Element entry = (Element)nodes.item(i);
                  if (! entry.getTagName().equals("entry"))
                     throw new Exception();
                  String entryName =
entry.getAttribute("name");
                  String entryNumber =
entry.getAttribute("number");
                  if (entryName.length() == 0 ||
entryNumber.length() == 0)
                     throw new Exception();
                  phoneBook.put(entryName,entryNumber);
               }
            }
         }
         catch (Exception e) {
            System.out.println("Error in read phone book
data file.");
            System.out.println("File name:  " +
dataFile.getAbsolutePath());
            System.out.println("This program cannot
continue.");
            System.exit(1);
         }
      }


      /* Read commands from the user and carry them out,
until the
       * user gives the "Exit from program" command.
       */

      Scanner in = new Scanner( System.in );
      boolean changed = false;  // Have any changes been
```

made to the directory?

```
      mainLoop: while (true) {
         System.out.println("\nSelect the action that you
want to perform:");
         System.out.println("   1.  Look up a phone
number.");
         System.out.println("   2.  Add or change a phone
number.");
         System.out.println("   3.  Remove an entry from
your phone directory.");
         System.out.println("   4.  List the entire phone
directory.");
         System.out.println("   5.  Exit from the
program.");
         System.out.println("Enter action number (1-5):
");
         int command;
         if ( in.hasNextInt() ) {
            command = in.nextInt();
            in.nextLine();
         }
         else {
            System.out.println("\nILLEGAL RESPONSE.  YOU
MUST ENTER A NUMBER.");
            in.nextLine();
            continue;
         }
         switch(command) {
         case 1:
            System.out.print("\nEnter the name whose number
you want to look up: ");
            name = in.nextLine().trim().toLowerCase();
            number = phoneBook.get(name);
            if (number == null)
               System.out.println("\nSORRY, NO NUMBER FOUND
FOR " + name);
            else
               System.out.println("\nNUMBER FOR " + name +
":  " + number);
            break;
         case 2:
            System.out.print("\nEnter the name: ");
            name = in.nextLine().trim().toLowerCase();
            if (name.length() == 0)
               System.out.println("\nNAME CANNOT BE
BLANK.");
            else if (name.indexOf('%') >= 0)
```

```
               System.out.println("\nNAME CANNOT CONTAIN
THE CHARACTER \"%\".");
            else {
               System.out.print("Enter phone number: ");
               number = in.nextLine().trim();
               if (number.length() == 0)
                  System.out.println("\nPHONE NUMBER CANNOT
BE BLANK.");
               else {
                  phoneBook.put(name,number);
                  changed = true;
               }
            }
            break;
         case 3:
            System.out.print("\nEnter the name whose entry
you want to remove: ");
            name = in.nextLine().trim().toLowerCase();
            number = phoneBook.get(name);
            if (number == null)
               System.out.println("\nSORRY, THERE IS NO
ENTRY FOR " + name);
            else {
               phoneBook.remove(name);
               changed = true;
               System.out.println("\nDIRECTORY ENTRY
REMOVED FOR " + name);
            }
            break;
         case 4:
            System.out.println("\nLIST OF ENTRIES IN YOUR
PHONE BOOK:\n");
            for ( Map.Entry<String,String> entry :
phoneBook.entrySet() )
               System.out.println("   " + entry.getKey() +
": " + entry.getValue() );
            break;
         case 5:
            System.out.println("\nExiting program.");
            break mainLoop;
         default:
            System.out.println("\nILLEGAL ACTION NUMBER.");
         }
      }


      /* Before ending the program, write the current
contents of the
```

```
       * phone directory, but only if some changes have
been made to
       * the directory.
       */

      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;
         }
         out.println("<?xml version=\"1.0\"?>");
         out.println("<phone_directory>");
         for ( Map.Entry<String,String> entry :
phoneBook.entrySet() ) {
            out.print("  <entry name='");
            out.print(entry.getKey());
            out.print("' number='");
            out.print(entry.getValue());
            out.println("'/>");
         }
         out.println("</phone_directory>");
         out.close();
         if (out.checkError())
            System.out.println("ERROR: Some error occurred
while writing data file.");
         else
            System.out.println("Done.");
      }

   }

}
```

Solution for Programming Exercise 11.6

Exercise 11.6:

The sample program Checkers.java from Subsection 7.6.3 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, showing that Load and Save buttons have been added:

It's a little tricky to restore the complete state of a game after reading the data from a file. 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 remaining program state correctly. This code assumes that you have introduced two new variables `saveButton` and `loadButton` of type *Button* 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.setDisable(true);
loadButton.setDisable(true);
saveButton.setDisable(false);
resignButton.setDisable(false);
if (currentPlayer == CheckersData.RED)
   message.setText("Game loaded -- it's RED's move.");
else
   message.setText("Game loaded -- it's BLACK's move.");
drawBoard();
```

Discussion

Many different formats could be used to represent the data for a checkers game in text form. In the program, the board data is an 8-by-8 two-dimensional array of integers. One way to represent the board would be to simply output 64 integers. However, the possible values in the array are given by constants `CheckersData.RED`, `CheckersData.BLACK`, `CheckersData.RED_KING`, `CheckersData.BLACK_KING`, and `CheckersData.EMPTY`, and it would be nice to use a representation that uses the **meaning** of these constants. I decided to represent the possible values with individual characters. An empty square is represented by '.'; a regular red piece by 'r'; a regular black piece by 'b'; a red king by 'R'; and a black king by 'B'. The contents of the 8-by-8 board are then represented by 8 lines with 8 characters on each line, separated by spaces. The spaces are there so that the characters can be read back individually from the file using a *Scanner*. I added a line at the beginning containing the name of the program "CheckersWithFiles"; this makes it easy to check that a file that the user selects for input is of the correct type. The last line of the file contains the

string "Red" if it is Red's turn to move or "Black" if it is Black's turn. Here is a sample file created by my program:

```
CheckersWithFiles
b . b . . . b .
. . . . . b . b
. . . . b . . .
. . . . . . . .
. . b . . . r .
. . . b . r . .
r . . . . . . .
. r . B . r . r
Red
```

I wrote methods `doSave()` and `doLoad()` to save and load files. They are modeled directly on the sample save and open methods that were presented in this chapter. The `doSave()` method uses a *PrintWriter* to write the file. The data is taken from the variables `board` and `currentPlayer`, as described in the exercise. This is very straightforward:

```
out.println("CheckersWithFiles"); // Identifies file as a
Checkers game.
for (int row = 0; row < 8; row++) {
   for (int col = 0; col < 8; col++) {
      int piece = board.pieceAt(row, col);  // Contents of
one square of board.
      switch (piece) {
         case CheckersData.EMPTY:      out.print(". ");
break;
         case CheckersData.RED:        out.print("r ");
break;
         case CheckersData.BLACK:      out.print("b ");
break;
         case CheckersData.RED_KING:   out.print("R ");
break;
         case CheckersData.BLACK_KING: out.print("B ");
break;
      }
   }
   out.println();
}
out.println(currentPlayer == CheckersData.RED ? "Red" :
"Black");
out.flush();
out.close();
```

```
if (out.checkError())
   throw new IOException("Some error occurred while saving
the file.");
```

Reading the data back in is not much harder. I used a *Scanner* named `in` to read the data from the file. The method `in.nextLine()` can be used to read the first line of text; `in.next()` is used to read the characters that represent the pieces on the board, as strings; and `in.next()` is also used to read the last word in the file, which will be "Red" or "Black". Remember that `in.next()` will skip over any spaces and end-of-lines before returning a non-blank token string; this makes it very easy to read the data format that I chose for the file. Here is the section of my `doLoad()` method that reads the data. It has a little more error-checking than you might have used in your own solution:

```
Scanner in;
try {
    in = new Scanner(selectedFile);
}
catch (Exception e) {
    Alert errorAlert = new Alert(Alert.AlertType.ERROR,
            "Sorry, but an error occurred while\n" +
                    "trying to open the file:\n" + e);
    errorAlert.showAndWait();
    return;
}
if (selectedFile == null)
    return;  // user canceled
try {
    CheckersData newBoard = new CheckersData();
    int newCurrentPlayer;
    String programNameFromFile = in.nextLine();
    if (! programNameFromFile.equals("CheckersWithFiles"))
        throw new Exception("Selected file does not
contain a checkers game.");
    for (int row = 0; row < 8; row++) {
        for (int col = 0; col < 8; col++) {
            String pieceCode = in.next();
            switch (pieceCode) {
            case ".": newBoard.setPieceAt(row, col,
CheckersData.EMPTY); break;
            case "r": newBoard.setPieceAt(row, col,
CheckersData.RED); break;
            case "b": newBoard.setPieceAt(row, col,
CheckersData.BLACK); break;
```

```
            case "R": newBoard.setPieceAt(row, col,
CheckersData.RED_KING); break;
            case "B": newBoard.setPieceAt(row, col,
CheckersData.BLACK_KING); break;
            default: throw new Exception("Illegal board
data found in file.");
            }
            if (row % 2 != col % 2 &&
                    newBoard.pieceAt(row, col) !=
CheckersData.EMPTY)
                throw new Exception("Illegal board data
found in file.");
        }
        in.nextLine();  // Discard rest of line -- not
really necessary.
    }
    String currentPlayerString = in.next();
    in.close(); // Done reading data from the file.
    if (currentPlayerString.equals("Red"))
        newCurrentPlayer = CheckersData.RED;
    else if (currentPlayerString.equals("Black"))
        newCurrentPlayer = CheckersData.BLACK;
    else
        throw new Exception("Illegal current player found
in file.");
    board = newBoard;  // Set up game with data read from
file.
    currentPlayer = newCurrentPlayer;
    legalMoves = board.getLegalMoves(currentPlayer);
    selectedRow = -1;
    gameInProgress = true;
    newGameButton.setDisable(true);
    loadButton.setDisable(true);
    saveButton.setDisable(false);
    resignButton.setDisable(false);
    if (currentPlayer == CheckersData.RED)
       message.setText("Game loaded -- it's RED's move.");
    else
       message.setText("Game loaded -- it's BLACK's
move.");
    drawBoard();
```

Note that after successfully reading data from the file, this method uses the code given in the exercise for restoring the state of the program to match the state of the game that has been read from the file.

Aside from writing the `doSave()` and `doLoad` methods, you must also add "Save Game" and "Load Game" buttons to the program, and you have to decide when they should be enabled and disabled. In my program, the user can save a game only when a game is actually in progress. The "Save" button is enabled as soon as a move is made in a game (`saveButton.setDisable(false)` is actually called every time a move is made, but only the first call matters). The "Save" button is disabled in the `gameOver()` method, which is always called when a game ends. As for the "Load" button, it is disabled when a game is in progress, and is enabled when a game ends. You can see all of the changes that I made in the solution below.

The Solution

The checkers program is very long. Changes from the original version are shown in red italic.

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.layout.Pane;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.input.MouseEvent;

import javafx.scene.control.Alert;
import javafx.stage.FileChooser;
import java.io.*;
import java.util.Scanner;

import java.util.ArrayList;


/**
 * This panel lets two users play checkers against each
other.
 * Red always starts the game.  If a player can jump on
```

```
 * Red always starts the game.  If a player can jump an
opponent's
 * piece, then the player must jump.  When a player can
make no more
 * moves, the game ends.
 *
 * The state of an ongoing game can be saved to a file at
any point,
 * and the resulting file can be loaded back into the
program to
 * restore the state of the game.  Load is only possible
when no
 * game is in progress.  "Save Game" and "Load Game"
buttons are
 * provided for saving and loading game state.
 */
public class CheckersWithFiles extends Application {

    public static void main(String[] args) {
        launch(args);
    }

    //-------------------------------------------------------
----------------

    CheckersBoard board; // A canvas on which a checker
board is drawn,
                         // defined by a static nested
subclass.  Much of
                         // the game logic is defined in
this class.


    private Button newGameButton;  // Button for starting a
new game.

    private Button resignButton;   // Button that a player
can use to end
                                   // the game by
resigning.

    private Button saveButton;     // For saving game state
to a file.
    private Button loadButton;     // For loading a saved
game.

    private Label message;  // Label for displaying
messages to the user
```

messages to the user.

```
    private Stage window;  // The program window (used as
parent for file dialog).

    /**
     * The constructor creates the Board (which in turn
creates and manages
     * the buttons and message label), adds all the
components, and sets
     * the bounds of the components.  A null layout is
used.  (This is
     * the only thing that is done in the main Checkers
class.)
     */
    public void start(Stage stage) {

        window = stage;

        /* Create the label that will show messages. */

        message = new Label("Click \"New Game\" to
begin.");
        message.setTextFill( Color.rgb(100,255,100) ); //
Light green.
        message.setFont( Font.font(null, FontWeight.BOLD,

18) );

        /* Create the buttons and the board.  The buttons
MUST be
         * created first, since they are used in the
CheckerBoard
         * constructor! */

        newGameButton = new Button("New Game");
        resignButton = new Button("Resign");
        saveButton = new Button("Save Game");
        loadButton = new Button("Load Game");

        board = new CheckersBoard(); // a subclass of
Canvas, defined below
        board.drawBoard();  // draws the content of the
checkerboard

        /* Set up ActionEvent handlers for the buttons and
a MousePressed handler
         * for the board.  The handlers call instance
methods in the board object. */
```

```
methods in the board object. */

        newGameButton.setOnAction( e -> board.doNewGame()
);
        resignButton.setOnAction( e -> board.doResign() );
        saveButton.setOnAction( e -> board.doSave() );
        loadButton.setOnAction( e -> board.doLoad() );
        board.setOnMousePressed( e -> board.mousePressed(e)
);

        /* Set the location of each child by calling its
relocate() method */

        board.relocate(20,20);
        newGameButton.relocate(370, 100);
        loadButton.relocate(370,150);
        saveButton.relocate(370,200);
        resignButton.relocate(370,250);
        message.relocate(20, 370);

        /* Set the sizes of the buttons.  For this to have
an effect, make
         * the butons "unmanaged."  If they are managed,
the Pane will set
         * their sizes. */


        resignButton.setManaged(false);
        resignButton.resize(100,30);
        loadButton.setManaged(false);
        loadButton.resize(100,30);
        saveButton.setManaged(false);
        saveButton.resize(100,30);
        newGameButton.setManaged(false);
        newGameButton.resize(100,30);

        /* Create the Pane and give it a preferred size.
If the
         * preferred size were not set, the unmanaged
buttons would
         * not be included in the Pane's computed preferred
size. */

        Pane root = new Pane();

        root.setPrefWidth(500);
        root.setPrefHeight(420);

        /* Add the child nodes to the Pane and set up the
```

```
        /* Add the child nodes to the Pane and set up the
rest of the GUI */

        root.getChildren().addAll(board, newGameButton,
loadButton,
                                                saveButton,
resignButton, message);
        root.setStyle("-fx-background-color: darkgreen; "
                            + "-fx-border-color: darkred; -
fx-border-width:3");
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setResizable(false);
        stage.setTitle("Checkers!");
        stage.show();

    } // end start()



    // --------------------  Nested Classes ---------------
----------------


    /**
     * 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.  Just 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
```

```
                // 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.



    /**
     * This canvas displays a 320-by-320 checkerboard
pattern with
     * a 2-pixel dark red border.  The canvas will be
exactly
     * 324-by-324 pixels. This class contains methods that
are
     * called in response to a mouse click on the canvas
and
     * in response to clicks on the New Game and Resign
buttons.
     * Note that the "New Game" and "Resign" buttons must
be
     * created before the Board constructor is called,
since
     * the constructor references the buttons (in the call
to doNewGame()).
     */
    private class CheckersBoard extends Canvas {

        CheckersData board; // The data for the checkers
board is kept here.
                            //    This board is also
responsible for generating
                            //    lists of legal moves.

        boolean gameInProgress; // Is a game currently in
progress?

        /* The next three variables are valid only when the
game is in progress. */

        int currentPlayer;      // Whose turn is it now?
The possible values
                                //    are CheckersData.RED
and CheckersData.BLACK
```

```
and CheckersData.BLACK.

        int selectedRow, selectedCol;   // If the current
player has selected a piece to
                                        //     move, these
give the row and column
                                        //     containing
that piece.  If no piece is
                                        //     yet
selected, then selectedRow is -1.

        CheckersMove[] legalMoves;  // An array containing
the legal moves for the
                                    //   current player.

        /**
         * Constructor.  Creates a CheckersData to
represent the
         * contents of the checkerboard, and calls
doNewGame to
         * start the first game.
         */
        CheckersBoard() {
            super(324,324);  // canvas is 324-by-324 pixels
            board = new CheckersData();
            doNewGame();
        }


        /**
         * Save the current state of the game to a user-
selected file.
         */
        void doSave() {
            FileChooser fileDialog = new FileChooser();
            fileDialog.setInitialFileName("checkers.txt");
            fileDialog.setInitialDirectory(new
File(System.getProperty("user.home")));
            fileDialog.setTitle("Select file where game
should be saved");
            File selectedFile =
fileDialog.showSaveDialog(window);
            if (selectedFile == null)
                return;  // user canceled
            PrintWriter out;
            try {
                FileWriter stream = new
FileWriter(selectedFile);
```

```
FileWriter(selectedFile);
                out = new PrintWriter( stream );
            }
            catch (Exception e) {
                Alert errorAlert = new
Alert(Alert.AlertType.ERROR,
                        "Sorry, but an error occurred
while\n" +
                                "trying to open the
file:\n" + e);
                errorAlert.showAndWait();
                return;
            }
            try {
                out.println("CheckersWithFiles"); //
Identifies file as a Checkers game.
                for (int row = 0; row < 8; row++) {
                    for (int col = 0; col < 8; col++) {
                        int piece = board.pieceAt(row,
col);
                        switch (piece) {
                        case CheckersData.EMPTY:
out.print(". ");  break;
                        case CheckersData.RED:
out.print("r ");  break;

                        case CheckersData.BLACK:
out.print("b ");  break;
                        case CheckersData.RED_KING:
out.print("R ");  break;
                        case CheckersData.BLACK_KING:
out.print("B ");  break;
                        }
                    }
                    out.println();
                }
                out.println(currentPlayer ==
CheckersData.RED ? "Red" : "Black");
                out.flush();
                out.close();
                if (out.checkError())
                    throw new IOException("Some error
occurred while saving the file.");
            }
            catch (Exception e) {
                Alert errorAlert = new
Alert(Alert.AlertType.ERROR,
```

```
Alert(Alert.AlertType.ERROR,
                        "Sorry, but an error occurred
while\n" +
                                "trying to write data to
the file:\n" + e);
                errorAlert.showAndWait();
            }
        }

        /**
         * Load a saved game from a file.
         */
        void doLoad() {
            FileChooser fileDialog = new FileChooser();
            fileDialog.setInitialDirectory(new
File(System.getProperty("user.home")));
            fileDialog.setTitle("Select file where game was
saved");
            File selectedFile =
fileDialog.showOpenDialog(window);
            if (selectedFile == null)
                return; // user canceled.
            Scanner in;
            try {
                in = new Scanner(selectedFile);

            }
            catch (Exception e) {
                Alert errorAlert = new
Alert(Alert.AlertType.ERROR,
                        "Sorry, but an error occurred
while\n" +
                                "trying to open the
file:\n" + e);
                errorAlert.showAndWait();
                return;
            }
            if (selectedFile == null)
                return;  // user canceled
            try {
                CheckersData newBoard = new CheckersData();
                int newCurrentPlayer;
                String programNameFromFile = in.nextLine();
                if (!
programNameFromFile.equals("CheckersWithFiles"))
                    throw new Exception("Selected file does
not contain a checkers game.");
```

```
not contain a checkers game. ");
                for (int row = 0; row < 8; row++) {
                    for (int col = 0; col < 8; col++) {
                        String pieceCode = in.next();
                        switch (pieceCode) {
                        case ".": newBoard.setPieceAt(row,
col, CheckersData.EMPTY); break;
                        case "r": newBoard.setPieceAt(row,
col, CheckersData.RED); break;
                        case "b": newBoard.setPieceAt(row,
col, CheckersData.BLACK); break;
                        case "R": newBoard.setPieceAt(row,
col, CheckersData.RED_KING); break;
                        case "B": newBoard.setPieceAt(row,
col, CheckersData.BLACK_KING); break;
                        default: throw new
Exception("Illegal board data found in file.");
                        }
                        if (row % 2 != col % 2 &&
                                newBoard.pieceAt(row, col)
!= CheckersData.EMPTY)
                            throw new Exception("Illegal
board data found in file.");
                    }
                    in.nextLine();  // Discard rest of line
-- not really necessary.
                }
                String currentPlayerString = in.next();
                in.close(); // Done reading data from the
file.
                if (currentPlayerString.equals("Red"))
                    newCurrentPlayer = CheckersData.RED;
                else if
(currentPlayerString.equals("Black"))
                    newCurrentPlayer = CheckersData.BLACK;
                else
                    throw new Exception("Illegal current
player found in file.");
                board = newBoard;  // Set up game with data
read from file.
                currentPlayer = newCurrentPlayer;
                legalMoves =
board.getLegalMoves(currentPlayer);
                selectedRow = -1;
                gameInProgress = true;
                newGameButton.setDisable(true);
```

```
                newGameButton.setDisable(true);
                loadButton.setDisable(true);
                saveButton.setDisable(false);
                resignButton.setDisable(false);
                if (currentPlayer == CheckersData.RED)
                    message.setText("Game loaded -- it's
RED's move.");
                else
                    message.setText("Game loaded -- it's
BLACK's move.");
                drawBoard();
            }
            catch (Exception e) {
                Alert errorAlert = new
Alert(Alert.AlertType.ERROR,
                        "Sorry, but an error occurred
while\n" +
                                "trying to read the
data:\n" + e);
                errorAlert.showAndWait();
            }
        }

        /**
         * Start a new game.  This method is called when

the Board is first
         * created and when the "New Game" button is
clicked.  Event handling
         * is set up in the start() method in the main
class.
         */
        void doNewGame() {
            if (gameInProgress == true) {
                    // This should not be possible, but it
doesn't hurt to check.
                message.setText("Finish the current game
first!");
                return;
            }
            board.setUpGame();   // Set up the pieces.
            currentPlayer = CheckersData.RED;   // RED
moves first.
            legalMoves =
board.getLegalMoves(CheckersData.RED);  // Get RED's legal
moves.
            selectedRow = -1;   // RED has not yet selected
a piece to move
```

```
a piece to move.
            message.setText("Red:  Make your move.");
            gameInProgress = true;
            newGameButton.setDisable(true);
            loadButton.setDisable(false);
            saveButton.setDisable(true);
            resignButton.setDisable(false);
            drawBoard();
        }

        /**
         * Current player resigns.  Game ends.  Opponent
wins.  This method is
         * called when the user clicks the "Resign" button.
Event handling is
         * set up in the start() method in the main class.
         */
        void doResign() {
            if (gameInProgress == false) {  // Should be
impossible.
                message.setText("There is no game in
progress!");
                return;
            }
            if (currentPlayer == CheckersData.RED)
                gameOver("RED resigns.  BLACK wins.");

            else
                gameOver("BLACK resigns.  RED wins.");
        }

        /**
         * The game ends.  The parameter, str, is displayed
as a message
         * to the user.  The states of the buttons are
adjusted so players
         * can start a new game.  This method is called
when the game
         * ends at any point in this class.
         */
        void gameOver(String str) {
            message.setText(str);
            newGameButton.setDisable(false);
            loadButton.setDisable(false);
            saveButton.setDisable(true);
            resignButton.setDisable(true);
            gameInProgress = false;
        }
```

```
        /**
         * This is called by mousePressed() when a player
clicks on the
         * square in the specified row and col.  It has
already been checked
         * that a game is, in fact, in progress.
         */
        void doClickSquare(int row, int col) {

            /* If the player clicked on one of the pieces
that the player
             can move, mark this row and col as selected
and return.  (This
             might change a previous selection.)  Reset the
message, in
             case it was previously displaying an error
message. */

            for (int i = 0; i < legalMoves.length; i++)
                if (legalMoves[i].fromRow == row &&
legalMoves[i].fromCol == col) {
                    selectedRow = row;
                    selectedCol = col;
                    if (currentPlayer == CheckersData.RED)
                        message.setText("RED:  Make your

move.");
                    else
                        message.setText("BLACK:  Make your
move.");
                    drawBoard();
                    return;
                }

            /* If no piece has been selected to be moved,
the user must first
             select a piece.  Show an error message and
return. */

            if (selectedRow < 0) {
                message.setText("Click the piece you want
to move.");
                return;
            }

            /* If the user clicked on a square where the
selected piece can be
             legally moved, then make the move and return
```

```
             legally moved, then make the move and return.
*/

            for (int i = 0; i < legalMoves.length; i++)
                if (legalMoves[i].fromRow == selectedRow &&
legalMoves[i].fromCol == selectedCol
                && legalMoves[i].toRow == row &&
legalMoves[i].toCol == col) {
                    doMakeMove(legalMoves[i]);
                    return;
                }

            /* If we get to this point, there is a piece
selected, and the square where
             the user just clicked is not one where that
piece can be legally moved.
             Show an error message. */

            message.setText("Click the square you want to
move to.");

        }  // end doClickSquare()

        /**
         * This is called when the current player has
chosen the specified
         * move.  Make the move, and then either end or
continue the game
         * appropriately.
         */
        void doMakeMove(CheckersMove move) {

            loadButton.setDisable(true);   // As soon as a
move is made, disable Load.
            saveButton.setDisable(false);  //     and
enable the Save.

            board.makeMove(move);

            /* If the move was a jump, it's possible that
the player has another
             jump.  Check for legal jumps starting from the
square that the player
             just moved to.  If there are any, the player
must jump.  The same
             player continues moving.
             */

            if (move.isJump()) {
```

```
            if (move.isJump()) {
                legalMoves =
board.getLegalJumpsFrom(currentPlayer,move.toRow,move.toCol
);
                if (legalMoves != null) {
                    if (currentPlayer == CheckersData.RED)
                        message.setText("RED:  You must
continue jumping.");
                    else
                        message.setText("BLACK:  You must
continue jumping.");
                    selectedRow = move.toRow;  // Since
only one piece can be moved, select it.
                    selectedCol = move.toCol;
                    drawBoard();
                    return;
                }
            }

            /* The current player's turn is ended, so
change to the other player.
             Get that player's legal moves.  If the player
has no legal moves,
             then the game ends. */

            if (currentPlayer == CheckersData.RED) {

                currentPlayer = CheckersData.BLACK;
                legalMoves =
board.getLegalMoves(currentPlayer);
                if (legalMoves == null)
                    gameOver("BLACK has no moves.  RED
wins.");
                else if (legalMoves[0].isJump())
                    message.setText("BLACK:  Make your
move.  You must jump.");
                else
                    message.setText("BLACK:  Make your
move.");
            }
            else {
                currentPlayer = CheckersData.RED;
                legalMoves =
board.getLegalMoves(currentPlayer);
                if (legalMoves == null)
                    gameOver("RED has no moves.  BLACK
wins.");
                else if (legalMoves[0].isJump())
                    message.setText("RED:  Make your move
```

```
                    message.setText("RED:  Make your move.
You must jump.");
                else
                    message.setText("RED:  Make your
move.");
            }

            /* Set selectedRow = -1 to record that the
player has not yet selected
             a piece to move. */

            selectedRow = -1;

            /* As a courtesy to the user, if all legal
moves use the same piece, then
             select that piece automatically so the user
won't have to click on it
             to select it. */

            if (legalMoves != null) {
                boolean sameStartSquare = true;
                for (int i = 1; i < legalMoves.length; i++)
                    if (legalMoves[i].fromRow !=
legalMoves[0].fromRow
                    || legalMoves[i].fromCol !=
legalMoves[0].fromCol) {
                        sameStartSquare = false;
                        break;
                    }
                if (sameStartSquare) {
                    selectedRow = legalMoves[0].fromRow;
                    selectedCol = legalMoves[0].fromCol;
                }
            }

            /* Make sure the board is redrawn in its new
state. */

            drawBoard();

        }  // end doMakeMove();

        /**
         * Draw a checkerboard pattern in gray and
lightGray.  Draw the
         * checkers.  If a game is in progress, highlight
the legal moves.
         */
        public void drawBoard() {
```

```
        public void drawBoard() {

            GraphicsContext g = getGraphicsContext2D();
            g.setFont( Font.font(18) );

            /* Draw a two-pixel black border around the
edges of the canvas. */

            g.setStroke(Color.DARKRED);
            g.setLineWidth(2);
            g.strokeRect(1, 1, 322, 322);

            /* Draw the squares of the checkerboard and the
checkers. */

            for (int row = 0; row < 8; row++) {
                for (int col = 0; col < 8; col++) {
                    if ( row % 2 == col % 2 )
                        g.setFill(Color.LIGHTGRAY);
                    else
                        g.setFill(Color.GRAY);
                    g.fillRect(2 + col*40, 2 + row*40, 40,
40);
                    switch (board.pieceAt(row,col)) {
                    case CheckersData.RED:
                        g.setFill(Color.RED);
                        g.fillOval(8 + col*40, 8 + row*40,
28, 28);
                        break;
                    case CheckersData.BLACK:
                        g.setFill(Color.BLACK);
                        g.fillOval(8 + col*40, 8 + row*40,
28, 28);
                        break;
                    case CheckersData.RED_KING:
                        g.setFill(Color.RED);
                        g.fillOval(8 + col*40, 8 + row*40,
28, 28);
                        g.setFill(Color.WHITE);
                        g.fillText("K", 15 + col*40, 29 +
row*40);
                        break;
                    case CheckersData.BLACK_KING:
                        g.setFill(Color.BLACK);
                        g.fillOval(8 + col*40, 8 + row*40,
28, 28);
                        g.setFill(Color.WHITE);
                        g.fillText("K", 15 + col*40, 29 +
row*40);
```

```
row 10;
                        break;
                    }
                }
            }

            /* If a game is in progress, highlight the
legal moves.   Note that legalMoves
             is never null while a game is in progress. */

            if (gameInProgress) {
                /* First, draw a 4-pixel cyan border around
the pieces that can be moved. */
                g.setStroke(Color.CYAN);
                g.setLineWidth(4);
                for (int i = 0; i < legalMoves.length; i++)
{
                    g.strokeRect(4 +
legalMoves[i].fromCol*40,
                                                    4 +
legalMoves[i].fromRow*40, 36, 36);
                }
                /* If a piece is selected for moving (i.e.
if selectedRow >= 0), then
                    draw a yellow border around that piece
and draw green borders
                    around each square that that piece can
be moved to. */
                if (selectedRow >= 0) {
                    g.setStroke(Color.YELLOW);
                    g.setLineWidth(4);
                    g.strokeRect(4 + selectedCol*40, 4 +
selectedRow*40, 36, 36);
                    g.setStroke(Color.LIME);
                    g.setLineWidth(4);
                    for (int i = 0; i < legalMoves.length;
i++) {
                        if (legalMoves[i].fromCol ==
selectedCol &&

legalMoves[i].fromRow == selectedRow) {
                            g.strokeRect(4 +
legalMoves[i].toCol*40,
                                                        4 +
legalMoves[i].toRow*40, 36, 36);
                        }
                    }
                }
            }
```

```
            ,

        }  // end drawBoard()

        /**
         * Respond to a user click on the board.  If no
game is in progress, show
         * an error message.  Otherwise, find the row and
column that the user
         * clicked and call doClickSquare() to handle it.
         */
        public void mousePressed(MouseEvent evt) {
            if (gameInProgress == false)
                message.setText("Click \"New Game\" to
start a new game.");
            else {
                int col = (int)((evt.getX() - 2) / 40);
                int row = (int)((evt.getY() - 2) / 40);
                if (col >= 0 && col < 8 && row >= 0 && row
< 8)
                    doClickSquare(row,col);
            }
        }


    }  // end class Board



    /**
     * An object of this class holds data about a game of
checkers.
     * It knows what kind of piece is on each square of the
checkerboard.
     * Note that RED moves "up" the board (i.e. row number
decreases)
     * while BLACK moves "down" the board (i.e. row number
increases).
     * Methods are provided to return lists of available
legal moves.
     */
    private static class CheckersData {

        /*  The following constants represent the possible
contents of a square
            on the board.  The constants RED and BLACK also
represent players
            in the game. */
```

```
        static final int
                    EMPTY = 0,
                    RED = 1,
                    RED_KING = 2,
                    BLACK = 3,
                    BLACK_KING = 4;

        int[][] board;  // board[r][c] is the contents of
row r, column c.

        /**
         * Constructor.  Create the board and set it up for
a new game.
         */
        CheckersData() {
            board = new int[8][8];
            setUpGame();
        }

        /**
         * Set up the board with checkers in position for
the beginning
         * of a game.  Note that checkers can only be found
in squares
         * that satisfy  row % 2 == col % 2.  At the start
of the game,
         * all such squares in the first three rows contain
black squares
         * and all such squares in the last three rows
contain red squares.
         */
        void setUpGame() {
            for (int row = 0; row < 8; row++) {
                for (int col = 0; col < 8; col++) {
                    if ( row % 2 == col % 2 ) {
                        if (row < 3)
                            board[row][col] = BLACK;
                        else if (row > 4)
                            board[row][col] = RED;
                        else
                            board[row][col] = EMPTY;
                    }
                    else {
                        board[row][col] = EMPTY;
                    }
                }
            }
        }  // end setUpGame()
```

```
        }  // end setupGame()

        /**
         * Return the contents of the square in the
specified row and column.
         */
        int pieceAt(int row, int col) {
            return board[row][col];
        }

        /**
         * Set the contents of the square in the specified
row and column.
         * piece must be one of the constants EMPTY, RED,
BLACK, RED_KING,
         * BLACK_KING.
         */
        void setPieceAt(int row, int col, int piece) {
            board[row][col] = piece;
        }

        /**
         * Make the specified move.  It is assumed that
move
         * is non-null and that the move it represents is
legal.
         */
        void makeMove(CheckersMove move) {
            makeMove(move.fromRow, move.fromCol,
move.toRow, move.toCol);
        }

        /**
         * 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];
            board[fromRow][fromCol] = EMPTY;
            if (fromRow - toRow == 2 || fromRow - toRow ==
```

```
            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;
            if (toRow == 7 && board[toRow][toCol] == BLACK)
                board[toRow][toCol] = BLACK_KING;
        }

        /**
         * 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) {

            if (player != RED && player != BLACK)
                return null;

            int playerKing;  // The constant representing a
King belonging to player.
            if (player == RED)
                playerKing = RED_KING;
            else
                playerKing = BLACK_KING;

            ArrayList<CheckersMove> moves = new
ArrayList<CheckersMove>();

            /*  First, check for any possible jumps.  Look
at each square on the board.
```

```
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 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

        /**
         * Return a list of the legal jumps that the
specified player can
         * make starting from the specified row and column.
If no such
```

```
if no such
         * jumps are possible, null is returned.  The logic
is similar
         * to the logic of the getLegalMoves() method.
         */
        CheckersMove[] getLegalJumpsFrom(int player, int
row, int col) {
            if (player != RED && player != BLACK)
                return null;
            int playerKing;  // The constant representing a
King belonging to player.
            if (player == RED)
                playerKing = RED_KING;
            else
                playerKing = BLACK_KING;
            ArrayList<CheckersMove> moves = new
ArrayList<CheckersMove>();
                   // The legal jumps will be stored in
this list.
            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 (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 getLegalMovesFrom()
```

```
        /**
         * This is called by the two previous 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) {

            if (r3 < 0 || r3 >= 8 || c3 < 0 || c3 >= 8)
                return false;  // (r3,c3) is off the board.

            if (board[r3][c3] != EMPTY)
                return false;  // (r3,c3) already contains
a piece.

            if (player == RED) {
                if (board[r1][c1] == RED && r3 > r1)
                    return false;  // Regular red piece can
only move up.
                if (board[r2][c2] != BLACK && board[r2][c2]
!= BLACK_KING)
                    return false;  // There is no black
piece to jump.
                return true;  // The jump is legal.
            }
            else {
                if (board[r1][c1] == BLACK && r3 < r1)
                    return false;  // Regular black piece
can only move downn.
                if (board[r2][c2] != RED && board[r2][c2]
!= RED_KING)
                    return false;  // There is no red piece
to jump.
                return true;  // The jump is legal.
            }

        }  // end canJump()

        /**
         * This is called by the getLegalMoves() method to
determine whether
```

```
determine whether
         * the player can legally move from (r1,c1) to
(r2,c2).  It is
         * assumed that (r1,r2) 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()

    } // end class CheckersData


} // end class CheckersWithFiles
```

Solution for Programming Exercise 12.1

Exercise 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 repeatedly call the `inc()` method in an object of type *Counter*. The object should be a shared global variable. 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).

Discussion

The necessary thread class is very simple to write. In my program, it's a static nested class that uses two global variables, which represent the counter and the number of times that the thread is to increment the counter:

```
static Counter counter;          // The counter that will
be incremented.
static int numberOfIncrements;   // Number of times each
thread will increment it.

static class IncrementerThread extends Thread {
    public void run() {
        for (int i = 0; i < numberOfIncrements; i++) {
            counter.inc();
        }
    }
}
```

Note that since the counter is a global static variable, it is used by all threads of type *IncrementerThread*. It is a shared resource. (`numberOfIncrements` is also shared, but its value never changes while the threads are running, so there is no question of race conditions for that variable.)

In the `main()` routine, after getting the number of threads and the number of increments from the user, we need to create the threads, start them, and wait for them to terminate. We have to do the part about starting all the threads before we get to the part about waiting for threads to terminate, so we need an array to hold the threads:

```
IncrementerThread[] workers = new
IncrementerThread[numberOfThreads];
counter = new Counter();
for (int i = 0; i < numberOfThreads; i++)
    workers[i] = new IncrementerThread();
for (int i = 0; i < numberOfThreads; i++)
    workers[i].start();

/* Wait for all threads to terminate. */

for (int i = 0; i < numberOfThreads; i++) {
    try {
        workers[i].join();
    }
    catch (InterruptedException e) {
```

```
        }
    }
```

(I hope that it's clear that you can't start a thread, wait for it to finish, start another thread, wait for it to finish, and so on. If you did that, you'd only have one thread running at any given time, and you would certainly not see any errors in the counter!)

My program actually runs in a loop, so the user can run several experiments without restarting the program every time.

In my own experiments, I saw a lot of errors when the number of increments per thread was one million, even with just two threads. For small numbers of increments, less than about 1000, I never saw any errors with two threads. (This might be because the threads didn't run long enough to actually get more than one running at a time.) I saw a few errors when I used a large number of threads with 1000 increments per thread.

Making `inc()` into a synchronized method completely solves the problem. If you try that, you should find that the program takes noticeably longer to run, because of the extra overhead associated with the synchronization. Simply declaring the variable `count` to be `volatile` does **not** fix the problem, since the race condition still exists. (In fact, curiously, I found that the number of errors increased when I made `count volatile`, possibly because accessing the volatile variable can cause a thread to block briefly and give another thread the chance to jump in.)

This exercise shows that synchronization errors can happen. In many programs, they would be very rare, but serious when they occur. Unfortunately, when trying to track down a bug that is due to a synchronization error, it might be very difficult to reproduce the error, since such errors depend on accidents of timing between two threads that are running independently. This makes it all the more important to analyze your program carefully and get it right when you are writing it in the first place, rather than count on your ability to debug it later.

The Solution

```
import java.util.Scanner;

/**
 * A demo program to see how hard it is to get an
 * error while using an unsynchronized counter with
 * several threads.  The program runs one or more
 * threads.  Each thread increments the counter a
 * specified number of times.  After all the threads
 * have completed, the value of the counter is
 * printed out so it can be compared to the correct
 * value.  The user specifies the number of threads
 * and the number of times each thread increments the
 * counter.  (To see an error, the number of increments
 * probably has to be pretty large.  Try 1000000.)
 */
public class UnsynchronizedCounterTest {

    /**
     * A class representing a counter with a method for
     * incrementing the counter.  No synchronization is
     * used, so this counter is not "thread-safe".
     */
    static class Counter {
        int count;
        void inc() {
            count = count+1;
        }
        int getCount() {
            return count;
        }
    }


    static Counter counter;          // The counter that
will be incremented.
    static int numberOfIncrements;   // Number of times
each thread will increment it.


    /**
     * The class that defines one of the threads.  The
thread
     * simply increments counter numberOfIncrements times,
in
     * a for loop.
     */
    static class IncrementerThread extends Thread {
        public void run() {
```

```
            for (int i = 0; i < numberOfIncrements; i++) {
                counter.inc();
            }
        }
    }


    /**
     * The main program runs in a loop until the user wants
to exit.
     * Each time through the loop, it runs one experiment.
It gets
     * the number of threads and the number of increments
per thread
     * from the user.  It creates and starts the threads,
and then
     * waits for them all to finish.  It prints the final
value of
     * the counter, as well as the expected value.  The
program ends
     * when the user enters a number less than or equal to
zero as
     * the number of threads.
     */
    public static void main(String[] args) {

        Scanner in = new Scanner(System.in);  // For
reading the user's inputs.

        while (true) {

            /* Get number of threads and number of
increments per thread
             * from the user.  Exit if number of threads is
<= 0. */

            System.out.println();
            System.out.print("How many threads do you want
to run (Enter 0 to end)? ");
            int numberOfThreads = in.nextInt();
            if (numberOfThreads <= 0)
                break;

            do {
                System.out.println();
                System.out.println("How many times should
each thread increment the counter? ");
                numberOfIncrements = in.nextInt();
```

```
                if (numberOfIncrements <= 0) {
                    System.out.println("Number of
increments must be positive.");
                }
            } while (numberOfIncrements <= 0);

            System.out.println();
            System.out.println("Using " + numberOfThreads +
" threads.");
            System.out.println("Each thread increments the
counter "
                                                +
numberOfIncrements + " times.");

            /* Create the threads and start them. */

            System.out.println();
            System.out.println("Working...");
            System.out.println();
            IncrementerThread[] workers = new
IncrementerThread[numberOfThreads];
            counter = new Counter();
            for (int i = 0; i < numberOfThreads; i++)
                workers[i] = new IncrementerThread();
            for (int i = 0; i < numberOfThreads; i++)
                workers[i].start();

            /* Wait for all threads to terminate. */

            for (int i = 0; i < numberOfThreads; i++) {
                try {
                    workers[i].join();
                }
                catch (InterruptedException e) {
                }
            }

            /* Display the results. */

            System.out.println("The final value of the
counter should be "
                                                          +
(numberOfIncrements*numberOfThreads));
            System.out.println("Actual final value of
counter is: " + counter.getCount());
            System.out.println();
            System.out.println();
```

```
        } // end while
    } // end main()

} // end class UnsynchronizedCounterTest
```

Solution for Programming Exercise 12.2

Exercise 12.2:

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. 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.

Discussion

Pseudocode for solving this problem without using threads is:

```
maxDivisors = 0    // Maximum number of divisors seen so
far
for N from 1 to 100000:
   Let count be the number of divisors of N
   if (count > maxDivisors):
      maxDivisors = count  // New maximum number of
divisors
      whichInt = N         // The integer that gave the
maximum
```

To turn this into a multithreaded program, we have to divide the integers between 1 and 100000 into groups and assign each group of integers to a thread. In its run method, each thread finds the maximum number of divisors for integers in its assigned group. This is very similar to what was done in the prime-counting program ThreadTest2.java. The only problem is

combining the final results of all the threads. In `ThreadTest2.java`, each thread counted some primes and recorded its results by calling a method

```
synchronized private static void addToTotal(int x)
```

We can do something similar here. The thread actually has two pieces of data to report, the maximum number of divisors that it found and the integer that had that many divisors. The data from the thread has to be combined with the overall information about the maximum found by any thread so far. So, the reporting method becomes:

```
/**
 * This method is called by a thread when it has completed
its computation,
 * to report the largest number of divisors that it found
in its assigned
 * range of integers.  The information is used to update
the variables
 * maxDivisorCount and intWithMaxDivisorCount.
 * @param maxCountFromThread largest divisor count in the
thread's assigned
 *          range
 * @param intWithMaxFromThread the integer that had the
maximum number
 *          of divisors
 */
synchronized private static void report(int
maxCountFromThread,
         int intWithMaxFromThread) {
   if (maxCountFromThread > maxDivisorCount) {
      maxDivisorCount = maxCountFromThread;
      intWithMaxDivisorCount = intWithMaxFromThread;
   }
}
```

The class that defines the threads, *CountDivisorsThread*, is very similar to the thread class from `ThreadTest2.java`. You can see it in the complete solution below.

In the program `ThreadTest2.java`, the number of integers to be processed was chosen to be evenly divisible by the number of threads. This made it easier to divide the integers among the threads, since every thread can be assigned exactly the same number of integers to process. For this program, I

decided to drop the assumption that the number of integers is divisible by the number of threads. This makes the program more general, but it's a little harder to decide how to assign the integers to the threads.

If there are `T` threads and `N` integers, then each thread gets at least `N/T` integers (where `N/T` means the integer quotient, discarding any remainder). But after assigning `N/T` integers to each thread, there might still be a few integers left over. (The number of leftovers is actually `N % T`.) In my program, I just add these extra integers onto the group of integers assigned to the last thread, so the last thread might have more integers to process than the other threads. I do this by making sure that the last number assigned to the last thread is 100000.)

```
int integersPerThread = 100000/numberOfThreads;
int start = 1;  // Starting point of the range of ints for
first thread.
int end = start + integersPerThread - 1;   // End point of
the range of ints.
for (int i = 0; i < numberOfThreads; i++) {
   if (i == numberOfThreads - 1) {
      end = 100000;  // Make sure that the last thread's
range goes all
                     // the way up to 100000.  Because of
rounding, this
                     // is not automatic.
   }
   worker[i] = new CountDivisorsThread( start, end );
   start = end+1;    // Determine the range of ints for
the NEXT thread.
   end = start + integersPerThread - 1;
}
```

As one final change, instead of using the literal number 100000 as the number of integers to be processed in my program, I use a constant named `MAX` to make it easier to change the range of integers that is considered.

The Solution

```
import java.util.Scanner;

/**
```

```
 * This program finds the number in the range 1 to some
maximum that has the
 * largest number of divisors.  It prints that number and
the number of divisors
 * that it has.  Note that there might be several numbers
that have the maximum
 * number of divisors.  Only one of them is output.
 *
 * The program's work is divided among one to ten threads.
The number
 * of threads is chosen by the user.
 */
public class CountDivisorsUsingThreads {

   /**
    * The upper limit of the range of integers that is to
be tested.
    */
   private final static int MAX = 100000;

   /**
    * The largest number of divisors found so far.  (Note:
This is
    * volatile since it is referenced in unsynchronized
code in the
    * countDivisorsWithThreads() method.)
    */
   private volatile static int maxDivisorCount = 0;

   /**
    * An integer that has the maximum number of divisors
seen so far.
    */
   private volatile static int intWithMaxDivisorCount;

   /**
    * This method is called by a thread when it has
completed its computation,
    * to report the largest number of divisors that it
found in its assigned
    * range of integers.  The information is used to update
the variables
    * maxDivisorCount and intWithMaxDivisorCount.
    * @param maxCountFromThread largest divisor count in
the thread's assigned
    *          range
    * @param intWithMaxFromThread the integer that had the
maximum number
```

```
    *          of divisors
    */
   synchronized private static void report(int
maxCountFromThread,
         int intWithMaxFromThread) {
      if (maxCountFromThread > maxDivisorCount) {
         maxDivisorCount = maxCountFromThread;
         intWithMaxDivisorCount = intWithMaxFromThread;
      }
   }

   /**
    * A thread belonging to this class counts the number of
divisors for all
    * the integers in an assigned range of integers.  The
range is specified
    * in the constructor.  The thread finds the integer in
the range that
    * has the largest number of divisors, and a number that
has that many
    * divisors.  At the end of its computation, the thread
reports its answer
    * by calling the report() method.
    */
   private static class CountDivisorsThread extends Thread
{
      int min, max;
      public CountDivisorsThread(int min, int max) {
         this.min = min;
         this.max = max;
      }
      public void run() {
//         System.out.println("Thread " + this + " testing
range " +
//               min + " to " + max);  // For testing.
//         long startTime = System.currentTimeMillis();
         int maxDivisors = 0;
         int whichInt = 0;
         for (int i = min; i < max; i++) {
            int divisors = countDivisors(i);
            if (divisors > maxDivisors) {
               maxDivisors = divisors;
               whichInt = i;
            }
         }
//         long elapsedTime = System.currentTimeMillis() -
startTime;
//         System.out.println("Thread " + this + " used " +
```

```
//                (elapsedTime/1000.0) + " seconds.");   //
for testing.
         report(maxDivisors,whichInt);
      }
   }

   /**
    * Finds the number in the range 1 to MAX that has the
largest number of
    * divisors, dividing the work among a specified number
of threads.
    */
   private static void countDivisorsWithThreads(int
numberOfThreads) {
      System.out.println("\nCounting divisors using " +
            numberOfThreads + " threads...");
      long startTime = System.currentTimeMillis();
      CountDivisorsThread[] worker = new
CountDivisorsThread[numberOfThreads];
      int integersPerThread = MAX/numberOfThreads;
      int start = 1;  // Starting point of the range of
ints for first thread.
      int end = start + integersPerThread - 1;   // End
point of the range of ints.
      for (int i = 0; i < numberOfThreads; i++) {
         if (i == numberOfThreads - 1) {
            end = MAX;  // Make sure that the last thread's
range goes all
                        // the way up to MAX.  Because of
rounding, this
                        // is not automatic.
         }
         worker[i] = new CountDivisorsThread( start, end );
         start = end+1;    // Determine the range of ints
for the NEXT thread.
         end = start + integersPerThread - 1;
      }
      maxDivisorCount = 0;
      for (int i = 0; i < numberOfThreads; i++)
         worker[i].start();
      for (int i = 0; i < numberOfThreads; i++) {
             // Wait for each worker thread to die, because
the results
             // are not complete until all threads have
completed and
             // reported their results.
         while (worker[i].isAlive()) {
            try {
```

```
                worker[i].join();
            }
            catch (InterruptedException e) {
            }
         }
      }
      long elapsedTime = System.currentTimeMillis() -
startTime;
      System.out.println("\nThe largest number of divisors
" +
            "for numbers between 1 and " + MAX + " is " +
maxDivisorCount);
      System.out.println("An integer with that many
divisors is " +
            intWithMaxDivisorCount);
      System.out.println("Total elapsed time:  " +
            (elapsedTime/1000.0) + " seconds.\n");
   }

   /**
    * Gets the number of threads from the user and calls
countDivisorsWithThreads()
    * to do the actual work.
    */
   public static void main(String[] args) {
      Scanner in = new Scanner(System.in);
      int numberOfThreads = 0;
      while (numberOfThreads < 1 || numberOfThreads > 10) {
         System.out.print("How many threads do you want to
use  (1 to 10) ?  ");
         numberOfThreads = in.nextInt();
         if (numberOfThreads < 1 || numberOfThreads > 10)
            System.out.println("Please enter a number from
1 to 10 !");
      }
      countDivisorsWithThreads(numberOfThreads);
   }

    /**
     * Finds the number of divisors of the integer N.  Note
that this method does
     * the counting in a stupid way, since it tests every
integer in the range
     * 1 to N to see whether it evenly divides N.
     */
   public static int countDivisors(int N) {
      int count = 0;
      for (int i = 1; i <= N ; i++) {
```

```
            if ( N % i == 0 )
               count ++;
         }
         return count;
      }

   }
```

Solution for Programming Exercise 12.3

Exercise 12.3:

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 results 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.

Discussion

I ran the solution to Exercise 12.2 on an old computer with four processors, using four threads. The four threads took widely varying amounts of time to complete: 2.6 seconds, 6.4 seconds, 12.9 seconds, and 20.4 seconds. The program ran for just over 20.4 seconds. For a lot of that time, there were only one or two threads running, leaving several processors idle. By using a thread pool as required by the current exercise, I was able to bring the total time for solving the problem down to 10.9 seconds. With the thread pool, the job was divided more evenly among the processors, and none of the processors were idle for very long.

In my solution, I decided to use a *ConcurrentLinkedQueue* for the task queue and a *LinkedBlockingQueue* for the result queue. Both of these classes are discussed in Section 12.3, and a *ConcurrentLinkedQueue* is used for the task queue in the sample program MultiprocessingDemo2.java from that section. (I could have used a blocking queue for the task queue too, as I'll discuss below.)

```
private static ConcurrentLinkedQueue<Task> taskQueue;
private static LinkedBlockingQueue<Result> resultQueue;
```

We need objects to put in the queues, to represent the tasks and the results. I decided to write a nested class to represent tasks and another nested class to represent results. A task is assigned a range of integers and has to find the largest number of divisors for any integer in that range. We also need to know which integer gave that maximum number. When the task is completed, the results from the task have to be added to the result queue. The classes that are neeed are very simple, and it is natural to define them as record classes (Section 7.4). Here are the *Task* and *Result* classes:

```
private record Task(int min, int max) {
    public void compute() {
        int maxDivisors = 0;
        int whichInt = 0;
        for (int i = min; i < max; i++) {
            int divisors = countDivisors(i);
            if (divisors > maxDivisors) {
                maxDivisors = divisors;
                whichInt = i;
            }
        }
        resultQueue.add( new Result(maxDivisors, whichInt)
```

```
);
    }
}

private record Result(int maxDivisorFromTask, int
intWithMaxFromTask) { }
```

The big idea when using a thread pool and load balancing is to divide up the overall problem into a fairly large number of fairly small (but not too small) subtasks. In my program, each task is assigned a range of 1000 integers to work on. Since there are 100000 integers in the overall problem, there will be 100 subtasks. The question of how big the subtasks should be does not have a definite answer, but this choice seemed to work well, since the work was divided pretty evenly among the processors.

Once the tasks have been created and the threads have been started, the main thread has to collect and process all the results from the tasks. As each task completes, it places its result into the result queue. A result can be removed from the queue by calling `resultQueue.take()`, which will block if the queue is empty until a result becomes available. So the for loop that processes results won't be able to complete until all the tasks have been completed and all the threads have terminated:

```
int maxDivisorCount = 0;         // Overall maximum found
by any task.
int intWithMaxDivisorCount = 0;  // Which integer gave
that maximum?
for (int i = 0; i < numberOfTasks; i++) {
    try {
        Result result = resultQueue.take();
        if (result.maxDivisorFromTask > maxDivisorCount) {
// new maximum.
            maxDivisorCount = result.maxDivisorFromTask;
            intWithMaxDivisorCount =
result.intWithMaxFromTask;
        }
    }
    catch (InterruptedException e) { // Won't happen in
this program!
    }
}
```

Once the for loop ends, the final answers, taking into account the results from every thread, are in the variables `maxDivisorCount` and `intWithMaxDivisorCount`.

My solution uses a *ConcurrentLinkedQueue* for the task queue. This is a little touchy, since we have to be careful not to start the threads until the tasks are in the queue. This is because the threads are programmed to terminate as soon as they see an empty queue:

```
private static class CountDivisorsThread extends Thread {
    public void run() {
        while (true) {
            Task task = taskQueue.poll();
            if (task == null)
                break; // queue is empty; terminate
            task.compute();
        }
    }
}
```

If the queue is empty when the thread starts, it will terminate immediately. Furthermore, terminating the threads after the job is done is fine in this program, but often when we use a thread pool, we want to keep the threads around to work on more than one job.

One solution is to use a *LinkedBlockingQueue* for the task queue, instead of a *ConcurrentLinkedQueue*. When a blocking queue is faced with an empty queue, it waits for an item to become available in the queue. We can then start the queue whenever we want, and the threads will wait, if necessary, for tasks to be added to the queue. If we had more jobs for the thread pool, the threads would just wait around between jobs and would become active again as soon as tasks from a new job are enqueued. Here is a version of the thread class that will work with a blocking queue of tasks:

```
private static class CountDivisorsThread extends Thread {
    CountDivisorsThread() {
        try {
            setDaemon(true);
        }
        catch (Exception e) {
```

```
        }
    }
    public void run() {
        while (true) {
            try {
                Task task = blockingTaskQueue.take();
                task.compute();
            }
            catch (InterruptedException e) {
            }
        }
    }
}
```

Note that I've made the thread into a daemon thread. This will allow the Java Virtual Machine to exit, even though the thread still exists, as long as all the non-daemon threads have exited. For this exercise, if the threads were not daemon threads, the program would not actually end after completing the job, since the non-daemon thread-pool threads would still be around. (However, you could still make the program end by calling `System.exit()`.)

The Solution

```
import java.util.concurrent.ConcurrentLinkedQueue;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.Scanner;

/**
 * This program finds the number in the range 1 to some
maximum that has the
 * largest number of divisors.  It prints that number and
the number of divisors
 * that it has.  Note that there might be several numbers
that have the maximum
 * number of divisors.  Only one of them is output.
 *
 * The program's work is divided into a large number of
tasks that are executed
 * by a thread pool.  Each task consists of finding the
maximum number of
 * divisors among a sequence of 1000 integers.
 */
public class CountDivisorsUsingThreadPool {
```

```
    /**
     * The upper limit of the range of integers that is to
be tested.
     * (This must be a fairly large multiple of 1000 for
the thread
     * pool load-balancing strategy to be effective.)
     */
    private final static int MAX = 100000;

    /**
     * A queue to hold the tasks.  Tasks are represented as
objects
     * of type Task, a nested class that is defined below.
Note
     * that queue operations must be synchronized because
the
     * queue is used by multiple threads.  A
ConcurrentLinkedQueue
     * handles synchronization internally.
     */
    private static ConcurrentLinkedQueue<Task> taskQueue;

    /**
     * A queue to hold the results from the tasks.  Results
     * are defined by the nested class, Result, which is
defined
     * below.  This is a blocking queue since the thread
     * that takes results from the queue should block when
     * the queue is empty until a result becomes available.
     * (Note:  The Task class could have been used to
represent
     * results as well; I am using a separate Result class
     * for clarity in this example.)
     */
    private static LinkedBlockingQueue<Result> resultQueue;


    /**
     * A class to represent the task of finding the number
in
     * a given range of integers that has the largest
number of
     * divisors.  The range is specified in the
constructor.
     * The task is executed when the compute() method is
     * called.  At the end of the compute() method, a
Result
     * object is created to represent the results from this
```

```
     * task, and the result object is added to resultQueue.
     */
    private record Task(int min, int max) {
        public void compute() {
            int maxDivisors = 0;
            int whichInt = 0;
            for (int i = min; i < max; i++) {
                int divisors = countDivisors(i);
                if (divisors > maxDivisors) {
                    maxDivisors = divisors;
                    whichInt = i;
                }
            }
            resultQueue.add( new Result(maxDivisors,
whichInt) );
        }
    }


    /**
     * A class to represent the result from one task.  The
     * result consists of the maximum number of divisors in
     * the range of integers assigned to that task, and the
     * integer in the range that gave the maximum number of
     * divisors.
     */
    private record Result(int maxDivisorFromTask, int
intWithMaxFromTask) { }


    /**
     * A thread belonging to this class counts the number
of divisors for all
     * the integers in an assigned range of integers.  The
range is specified
     * in the constructor.  The thread finds the integer in
the range that
     * has the largest number of divisors, and a number
that has that many
     * divisors.  At the end of its computation, the thread
reports its answer
     * by calling the report() method.
     */
    private static class CountDivisorsThread extends Thread
{
        public void run() {
            while (true) {
                Task task = taskQueue.poll();
```

```
                if (task == null)
                    break;
                task.compute();
            }
        }
    }


    /**
     * Finds the number in the range 1 to MAX that has the
largest number of
     * divisors, dividing the work into tasks that will be
executed by threads
     * in a thread pool.  This method creates the task and
result queues.
     * It adds all the tasks to the task queue.  Then it
creates the threads
     * for the thread pool and starts them.  (Note that
this must be done
     * AFTER all the tasks are in the task queue, since the
threads exit
     * when they see an empty queue.)  Finally, it reads
results from
     * the result queue and combines them to get the
overall answer.
     * @param numberOfThreads the number of threads in the
thread pool.
     */
    private static void countDivisorsWithThreads(int
numberOfThreads) {

        System.out.println("\nCounting divisors using " +
                                                numberOfThreads
+ " threads...");

        /* Create the queues and the thread pool, but don't
start
         * the threads yet. */

        long startTime = System.currentTimeMillis();
        resultQueue = new LinkedBlockingQueue<Result>();

        taskQueue = new ConcurrentLinkedQueue<Task>();
        CountDivisorsThread[] workers = new
CountDivisorsThread[numberOfThreads];
        for (int i = 0; i < workers.length; i++)
            workers[i] = new CountDivisorsThread();
```

```
        /* Create the tasks and add them to the task queue.
Each
         * task consists of a range of 1000 integers, so
the number of
         * tasks is (MAX+999)/1000.  (The "+999"  gives the
correct number
         * of tasks when MAX is not an exact multiple of
1000.  The last
         * task in that case will consist of the last
(MAX%1000)) ints. */

        int numberOfTasks = (MAX + 999) / 1000;
        for (int i = 0; i < numberOfTasks; i++) {
            int start = i*1000 + 1;
            int end = (i+1)*1000;
            if (end > MAX)
                end = MAX;
            //System.out.println(start + " " + end);  //
for testing
            taskQueue.add( new Task(start,end) );
        }

        /* Now that the tasks are in the task queue, start
the threads. */

        for (int i = 0; i < numberOfThreads; i++)
            workers[i].start();

        /* The threads will execute the tasks and results
will be placed
         * into the result queue.  This method now goes on
to read all
         * the results from the result queue and combine
them to give
         * the overall answer. */

        int maxDivisorCount = 0;         // Over maximum
found by any task.
        int intWithMaxDivisorCount = 0;  // Which integer
gave that maximum?
        for (int i = 0; i < numberOfTasks; i++) {
            try {
                Result result = resultQueue.take();
                if (result.maxDivisorFromTask >
maxDivisorCount) { // new maximum.
                    maxDivisorCount =
result.maxDivisorFromTask;
                    intWithMaxDivisorCount =
```

```
result.intWithMaxFromTask;
                }
            }
            catch (InterruptedException e) {
            }
        }

        /* Report the results. */

        long elapsedTime = System.currentTimeMillis() -
startTime;
        System.out.println("\nThe largest number of
divisors " +
                "for numbers between 1 and " + MAX + " is "
+ maxDivisorCount);
        System.out.println("An integer with that many
divisors is " +
                intWithMaxDivisorCount);
        System.out.println("Total elapsed time:  " +
                (elapsedTime/1000.0) + " seconds.\n");

    } // end countDivisorsWithThreads()


    /**
     * The main() routine just gets the number of threads
from the user and
     * calls countDivisorsWithThreads() to do the actual
work.
     */
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int numberOfThreads = 0;
        while (numberOfThreads < 1 || numberOfThreads > 10)
{
            System.out.print("How many threads do you want
to use  (1 to 10) ?  ");
            numberOfThreads = in.nextInt();
            if (numberOfThreads < 1 || numberOfThreads >
10)
                System.out.println("Please enter a number
from 1 to 10 !");
        }
        countDivisorsWithThreads(numberOfThreads);
    }


    /**
```

```
     * Finds the number of divisors of the integer N.  Note
that this method does
     * the counting in a stupid way, since it tests every
integer in the range
     * 1 to N to see whether it evenly divides N.
     */
    private static int countDivisors(int N) {
        int count = 0;
        for (int i = 1; i <= N ; i++) {
            if ( N % i == 0 )
                count ++;
        }
        return count;
    }

} // end CountDivisorsUsingThreadPool
```

Solution for Programming Exercise 12.4

Exercise 12.4:

In previous exercise, you used a thread pool and a queue of tasks to find the integer in the range 1 to 100000 that has the largest number of divisors. Subsection 12.3.4 discusses a higher-level approach that uses an *ExecutorService*. Write one more program to solve the problem, this time using an *ExecutorService* and *Futures*. The program should still break up the computation into a fairly large number of fairly small tasks, and it should still print out the largest number of divisors and the integer that has that number of divisors.

(There is yet another way to solve the same problem: the stream API from Section 10.6. My on-line solution of this exercise also discusses how to use streams to solve the problem.)

Discussion

My solution to Exercise 12.3 used nested classes *Task* and *Result* to represent one of the subtasks in the problem and the result from a subclass. For this problem, we want to submit the tasks to an *ExecutorService*, so we need tasks that implement the *Callable* interface. The tasks compute outputs of type *Result*, so in fact, *Task* implements `Callable<Result>`. Here is the new *Task* class, with changes shown in red italic:

```
/**
 * A class to represent the task of finding the number in
 * a given range of integers that has the largest number
of
 * divisors.  The range is specified in the constructor.
 * The task is executed when the call() method is
 * called.  At the end of the call() method, a Result
 * object is created to represent the results from this
 * task, and the result object is returned as the value
```

```
 * of call().
 */
private record Task(int min, int max) implements
Callable<Result>{
    public Result call() {
        int maxDivisors = 0;
        int whichInt = 0;
        for (int i = min; i < max; i++) {
            int divisors = countDivisors(i);
            if (divisors > maxDivisors) {
                maxDivisors = divisors;
                whichInt = i;
            }
        }
        return new Result(maxDivisors,whichInt);
    }
}
```

Following the example of ThreadTest4.java, we can perform the computation by creating an *ExecutorService* and submitting the tasks to it. When a task it submitted, an object of type `Future<Result>` is returned, representing the future result of the computation. The program needs to save those *Futures* in a list:

```
ExecutorService executor =
Executors.newFixedThreadPool(numberOfThreads);
ArrayList<Future<Result>> results = new ArrayList<>();

int numberOfTasks = (MAX + 999) / 1000;
for (int i = 0; i < numberOfTasks; i++) {
    int start = i*1000 + 1;
    int end = (i+1)*1000;
    if (end > MAX)
        end = MAX;
    //System.out.println(start + " " + end);  // for
testing
    Future<Result> res = executor.submit( new
Task(start,end) );
    results.add(res);
}
```

After submitting all the tasks, the program can get the results from the *Futures*. Note that calling a Future's `get()` method will block until the result is actually available (or an error occurs):

```
int maxDivisorCount = 0;          // Overall maximum found
by any task.
int intWithMaxDivisorCount = 0;   // Which integer gave
that maximum?
for (Future<Result> res : results) {
    try {
        Result result = res.get();
        if (result.maxDivisorFromTask > maxDivisorCount) {
// new maximum.
            maxDivisorCount = result.maxDivisorFromTask;
            intWithMaxDivisorCount =
result.intWithMaxFromTask;
        }
    }
    catch (Exception e) {
        System.out.println("An unexpected error occurred!
Error:");
        System.out.println(e);
        System.exit(1);
    }
}
```

In my testing, this version of the solution to the divisors program took about the same amount of time as the solution that implemented thread pools directly.

As a final note, `executor.shutdown()` needs to be called at some point after all the tasks have been submitted. Otherwise, the program won't actually end when `main()` finishes, since the non-daemon threads in the executor's thread pool will still exist. Alternatively, `System.exit()` could be called to definitively end the program.

There is yet another way to solve the "count divisors" problem: using the stream API. The same *Task* and *Result* classes that were used in the *ExectorService* approach can also be used with streams (although *Task* no longer needs to implement *Callable*).

The algorithm goes like this: Create an `ArrayList<Task>` containing all of the tasks that make up the problem. Obtain a `parallelStream` from that list. Apply a `mapping` operation to the stream, where each task is mapped to the result that is output from that task. (This is where all of the work of counting

is done.) This converts the stream of *Tasks* into a stream of *Results*. Finally, we need to find the result with the largest number of divisors. That can be done by applying the `max()` operator to the stream. The `max()` operator takes a parameter that is the *Comparator* that it will use to compare two objects from the stream. So, starting from the *ArrayList*, `tasks`, the computation can be done as follows:

```
tasks.parallelStream()
     .map( task -> task.call() )
     .max( (r1,r2) -> r1.maxDivisorFromTask -
r2.maxDivisorFromTask );
```

This computation returns the result that contains a *Result* with the largest `maxDivisorFromTask`. Unfortunately, it returns the answer as a value of type `Optional<Result>` (Subsection 10.6.3). If `opt` is the *Optional* value, then the actual value is retrieved by calling `opt.get()`, which will throw an exception if the value does not exist. The reason `max()` returns an *Optional* is that an empty stream does not have a maximum, so no value will be present in that case. For this problem, we know that the maximum exists, so we can call `get()` without checking whether the value is present. Here is a `main()` program that uses the stream API to solve the divisors problem:

```
public static void main(String[] args) {
    long startTime = System.currentTimeMillis();
    ArrayList<Task> tasks = new ArrayList<>();
    int numberOfTasks = (MAX + 999) / 1000;
    for (int i = 0; i < numberOfTasks; i++) {
        int start = i*1000 + 1;
        int end = (i+1)*1000;
        if (end > MAX)
            end = MAX;
        tasks.add( new Task(start,end) );
    }
    Optional<Result> max;
    max = tasks.parallelStream()
           .map( task -> task.call() )
           .max( (r1,r2) -> r1.maxDivisorFromTask -
r2.maxDivisorFromTask);
    Result bestRes = max.get();
    long elapsedTime = System.currentTimeMillis() -
startTime;
    System.out.println("\nThe largest number of divisors "
+
```

```
            "for numbers between 1 and " + MAX +
            " is " + bestRes.maxDivisorFromTask);
    System.out.println("An integer with that many divisors
is " +
            bestRes.intWithMaxFromTask);
    System.out.println("Total elapsed time:  " +
            (elapsedTime/1000.0) + " seconds.\n");
}
```

In my testing, this version of the solution took about 12% more time than the *ExecutorService* solution or the version that implemented a thread pool directly.

The Solution

Here is a solution, with changes from the solution to the previous exercise shown in red italic:

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Callable;
import java.util.concurrent.Future;
import java.util.ArrayList;
import java.util.Scanner;

/**
 * This program finds the number in the range 1 to some
maximum that has the
 * largest number of divisors.  It prints that number and
the number of divisors
 * that it has.  Note that there might be several numbers
that have the maximum
 * number of divisors.  Only one of them is output.
 *
 * The program's work is divided into a large number of
tasks that are executed
 * by an ExecutorService.  Each task consists of finding
the maximum number of
 * divisors among a sequence of 1000 integers.
 */
public class CountDivisorsUsingExecutor {

    /**
```

```
     * The upper limit of the range of integers that is to
be tested.
     * (This must be a fairly large multiple of 1000 for
the thread
     * pool load-balancing strategy to be effective.)
     */
    private final static int MAX = 100000;


    /**
     * A class to represent the result from one task.  The
     * result consists of the maximum number of divisors in
     * the range of integers assigned to that task, and the
     * integer in the range that gave the maximum number of
     * divisors.
     */
    private record Result(int maxDivisorFromTask, int
intWithMaxFromTask) { }


   /**
     * A class to represent the task of finding the number
in
     * a given range of integers that has the largest
number of
     * divisors.  The range is specified in the
constructor.
     * The task is executed when the call() method is
     * called.  At the end of the call() method, a Result
     * object is created to represent the results from this
     * task, and the result object is returned as the value
     * of call().
     */
    private record Task(int min, int max) implements
Callable<Result>{
        public Result call() {
            int maxDivisors = 0;
            int whichInt = 0;
            for (int i = min; i < max; i++) {
                int divisors = countDivisors(i);
                if (divisors > maxDivisors) {
                    maxDivisors = divisors;
                    whichInt = i;
                }
            }
            return new Result(maxDivisors,whichInt);
        }
```

```
    }


    /**
     * Finds the number in the range 1 to MAX that has the
largest number of
     * divisors, dividing the work into tasks that will be
submitted to an
     * ExecutorService.  The Futures that are returned when
the tasks are
     * submitted are placed into an ArrayList.  The results
from those Futures
     * are combined to produce the final output.
     * @param numberOfThreads the number of threads to be
used by the executor
     */
    private static void countDivisorsWithExecutor(int
numberOfThreads) {

        System.out.println("\nCounting divisors using " +
                                                numberOfThreads
+ " threads...");

        /* Create the ExecutorService and an ArrayList to
hold the Futures. */

        long startTime = System.currentTimeMillis();
        ExecutorService executor =
Executors.newFixedThreadPool(numberOfThreads);

        ArrayList<Future<Result>> results = new ArrayList<>
();

        /* Create the tasks and add them to the executor.
Each
         * task consists of a range of 1000 integers, so
the number of
         * tasks is (MAX+999)/1000.  (The "+999"  gives the
correct number
         * of tasks when MAX is not an exact multiple of
1000.  The last
         * task in that case will consist of the last
(MAX%1000)) ints. */

        int numberOfTasks = (MAX + 999) / 1000;
        for (int i = 0; i < numberOfTasks; i++) {
            int start = i*1000 + 1;
```

```
            int end = (i+1)*1000;
            if (end > MAX)
                end = MAX;
            //System.out.println(start + " " + end);  //
for testing
            Future<Result> res = executor.submit( new
Task(start,end) );
            results.add(res);
        }

        /* As the executor executes the tasks, results
become available
         * in the Futures that are stored in the ArrayList.
Get the
         * results and combine them to produce the final
output.
         * Note that each call to res.get() blocks, if
necessary,
         * until the result is available. */

        int maxDivisorCount = 0;         // Over maximum
found by any task.
        int intWithMaxDivisorCount = 0;  // Which integer
gave that maximum?
        for (Future<Result> res : results) {
            try {
                Result result = res.get();
                if (result.maxDivisorFromTask >
maxDivisorCount) { // new maximum.
                    maxDivisorCount =
result.maxDivisorFromTask;
                    intWithMaxDivisorCount =
result.intWithMaxFromTask;
                }
            }
            catch (Exception e) {
                System.out.println("An unexpected error
occurred! Error:");
                System.out.println(e);
                System.exit(1);
            }
        }

        /* Report the results. */

        long elapsedTime = System.currentTimeMillis() -
startTime;
```

```
        System.out.println("\nThe largest number of
divisors " +
                "for numbers between 1 and " + MAX + " is "
+ maxDivisorCount);
        System.out.println("An integer with that many
divisors is " +
                intWithMaxDivisorCount);
        System.out.println("Total elapsed time:  " +
                (elapsedTime/1000.0) + " seconds.\n");

        executor.shutdown(); // Needed since otherwise the
threads in the
                             // ExecutorService will stop
the Java Virtual
                             // Machine from shutting down
normally.

    } // end countDivisorsWithExecutor()


    /**
     * The main() routine just gets the number of threads
from the user and
     * calls countDivisorsWithThreads() to do the actual
work.
     */
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int numberOfThreads = 0;
        while (numberOfThreads < 1 || numberOfThreads > 10)
{
            System.out.print("How many threads do you want
to use  (1 to 10) ?  ");
            numberOfThreads = in.nextInt();
            if (numberOfThreads < 1 || numberOfThreads >
10)
                System.out.println("Please enter a number
from 1 to 10 !");
        }
        countDivisorsWithExecutor(numberOfThreads);
    }


    /**
     * Finds the number of divisors of the integer N.  Note
that this method does
     * the counting in a stupid way, since it tests every
```

```
integer in the range
     * 1 to N to see whether it evenly divides N.
     */
    private static int countDivisors(int N) {
        int count = 0;
        for (int i = 1; i <= N ; i++) {
            if ( N % i == 0 )
                count ++;
        }
        return count;
    }

} // end CountDivisorsUsingExecutor
```

And here is a version that uses the stream API:

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Callable;
import java.util.concurrent.Future;
import java.util.ArrayList;
import java.util.Scanner;
import java.util.Optional;

/**
 * This program finds the number in the range 1 to some
maximum that has the
 * largest number of divisors.  It prints that number and
the number of divisors
 * that it has.  Note that there might be several numbers
that have the maximum
 * number of divisors.  Only one of them is output.
 *
 * The version of the program uses the stream API.
 */
public class CountDivisorsUsingStreamAPI {

    /**
     * The upper limit of the range of integers that is to
be tested.
     * (This must be a fairly large multiple of 1000 for
the thread
     * pool load-balancing strategy to be effective.)
     */
    private final static int MAX = 100000;
```

```
    /**
     * A class to represent the result from one task.  The
     * result consists of the maximum number of divisors in
     * the range of integers assigned to that task, and the
     * integer in the range that gave the maximum number of
     * divisors.
     */
    private record Result(int maxDivisorFromTask, int
intWithMaxFromTask) { }


   /**
     * A class to represent the task of finding the number
in
     * a given range of integers that has the largest
number of
     * divisors.  The range is specified in the
constructor.
     * The task is executed when the call() method is
     * called.  At the end of the call() method, a Result
     * object is created to represent the results from this
     * task, and the result object is returned as the value
     * of call().
     */
    private record Task(int min, int max) implements
Callable<Result>{
        public Result call() {
            int maxDivisors = 0;
            int whichInt = 0;
            for (int i = min; i < max; i++) {
                int divisors = countDivisors(i);
                if (divisors > maxDivisors) {
                    maxDivisors = divisors;
                    whichInt = i;
                }
            }
            return new Result(maxDivisors,whichInt);
        }
    }


    /**
     * The main() routine uses the stream API to find the
number in the range
     * 0 to MAX that has the largest number of divisors.
     */
    public static void main(String[] args) {
```

```
        long startTime = System.currentTimeMillis();
        ArrayList<Task> tasks = new ArrayList<>();
        int numberOfTasks = (MAX + 999) / 1000;
        for (int i = 0; i < numberOfTasks; i++) {
            int start = i*1000 + 1;
            int end = (i+1)*1000;
            if (end > MAX)
                end = MAX;
            tasks.add( new Task(start,end) );
        }
        Optional<Result> max;
        max = tasks.parallelStream()
               .map( task -> task.call() )
               .max( (r1,r2) -> r1.maxDivisorFromTask -
r2.maxDivisorFromTask);
        Result bestRes = max.get();
        long elapsedTime = System.currentTimeMillis() -
startTime;
        System.out.println("\nThe largest number of
divisors " +
                "for numbers between 1 and " + MAX +
                " is " + bestRes.maxDivisorFromTask);
        System.out.println("An integer with that many
divisors is " +
                bestRes.intWithMaxFromTask);
        System.out.println("Total elapsed time:  " +
                (elapsedTime/1000.0) + " seconds.\n");
    }


    /**
     * Finds the number of divisors of the integer N.  Note
that this method does
     * the counting in a stupid way, since it tests every
integer in the range
     * 1 to N to see whether it evenly divides N.
     */
    private static int countDivisors(int N) {
        int count = 0;
        for (int i = 1; i <= N ; i++) {
            if ( N % i == 0 )
                count ++;
        }
        return count;
    }

} // end CountDivisorsUsingStreamAPI
```

Solution for Programming Exercise 12.5

Exercise 12.5:

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.

Discussion

This is not a difficult exercise, given the model in `DateServerWithThreads.java`. My solution to Exercise 11.3, `FileServer,java`, already has a method

```
private static void handleConnection(File directory,
Socket connection)
```

to handle a connection with one client. The connection in this method is represented by a *Socket*. The *File* in this method represents the directory that contains the file server's files; it is set at the beginning of the program and does not change after that. The only thing that we have to send from the main program to the connection-handling threads is the socket that represents a given connection. So, the queue can be a queue of *Sockets*:

```
private static ArrayBlockingQueue<Socket> connectionQueue;
```

A connection-handling thread runs in an infinite loop in which it takes a socket from this queue and uses it to communicate with the client. I could have moved the code from the `handleConnection()` method into the run method of the thread, but I decided it would be easier to have the `run()` method call the method that already exists. I also need to pass the directory to the `handleConnection()` method, so the thread needs some way of knowing the directory. I pass the directory to the thread in the thread's constructor. (An alternative would have been to make it a global variable, as I did for the queue.) Here is the class that defines the connection-handling threads:

```
private static class ConnectionHandler extends Thread {
    File directory;  // The directory that contains the
files
                     // that are made available on this
server.
    ConnectionHandler(File directory) {
        this.directory = directory;
        setDaemon(true);
    }
    public void run() {
        while (true) {
            try {
                Socket connection =
connectionQueue.take();
                handleConnection(directory,connection);
            }
            catch (Exception e) {
            }
        }
    }
}
```

The main program creates and starts several threads belonging to this class, before it starts listening for connection requests. Other changes are straightforward. You can see the details in the complete solution, given below.

The Solution

Changes from the solution of Exercise 11.3 are shown in red italic:

```
import java.net.*;
import java.io.*;
import java.util.Scanner;
import java.util.concurrent.ArrayBlockingQueue;

/**
 * This program is a very simple network file server.  The
 * server has a list of available text files that can be
 * downloaded by the client.  The client can also download
 * the list of files.  When the connection is opened, the
 * client sends one of two possible commands to the server:
 * "index" or "get <file-name>".  The server replies to
 * the first command by sending the list of available
files.
 * It responds to the second with a one-line message,
 * either "ok" or "error".  If the message is "ok", it is
 * followed by the contents of the file with the specified
 * name.  The "error" message indicates that the specified
 * file does not exist on the server. (The server can also
 * respond with the message "unknown command" if the
command
 * it reads is not one of the two possible legal commands.)
 *
 * The server program requires a command-line parameter
 * that specifies the directory that contains the files
 * that the server can serve.  The files should all be
 * text files, but this is not checked.  Also, the server
 * must have permission to read all the files.
 *
 * This version of the program defines a multithreaded
 * server that uses a thread pool.  The threads handle
 * all communication with the clients.  The main program
 * simply accepts connections and puts them into a queue.
 * The connection-handling threads in the thread pool
remove
 * connections from the queue as they become available.
 */
public class ThreadedFileServer {

    static final int LISTENING_PORT = 3210;

    /**
     * The number of threads in the thread pool.
     */
    private static final int THREAD_POOL_SIZE = 10;

    /**
```

```
     * The length of the ArrayBlockingQueue of connections.
     * This should not be too big, since connections in the
     * queue are waiting for service and hopefully won't
     * spend too long in the queue.
     */
    private static final int CONNECTION_QUEUE_SIZE = 5;

    /**
     * The queue that is used to send connections from the
     * main program to the connection-handling threads.
     * A connection is represented by a connected Socket.
     */
    private static ArrayBlockingQueue<Socket>
connectionQueue;


    /**
     * Main program creates the thread pool, then opens a
     * server socket to listen for connection requests.
     */
    public static void main(String[] args) {

        File directory;        // The directory from which
the server
                               //    gets the files that it
serves.

        ServerSocket listener; // Listens for connection
requests.

        Socket connection;     // A socket for
communicating with a client.


        /* Check that there is a command-line argument.
           If not, print a usage message and end. */

        if (args.length == 0) {
            System.out.println("Usage:  java FileServer
<directory>");
            return;
        }

        /* Get the directory name from the command line,
and make
           it into a file object.  Check that the file
exists and
```

```
           is in fact a directory. */

        directory = new File(args[0]);
        if ( ! directory.exists() ) {
            System.out.println("Specified directory does
not exist.");
            return;
        }
        if (! directory.isDirectory() ) {
            System.out.println("The specified file is not a
directory.");
            return;
        }

        /* Create the connection queue.  We want to do this
before
         * creating the threads, which need to use the
queue. */

        connectionQueue = new ArrayBlockingQueue<Socket>
(CONNECTION_QUEUE_SIZE);

        /* Create the thread pool and start the threads.
The directory
         * that contains the files is passed to each thread
as a
         * parameter to its constructor.  Note that there
is no
         * need to keep references to the threads, since we
don't have
         * to do anything with them in this program after
they have
         * been started.*/

        for (int i = 0; i < THREAD_POOL_SIZE; i++) {
            ConnectionHandler worker = new
ConnectionHandler(directory);
            worker.start();
        }

        /* Listen for connection requests from clients.
For each
         * connection, add the connected socket to the
connection
         * queue.  The server runs until the program is
terminated,
```

```
         * for example by a CONTROL-C. */

        try {
            listener = new ServerSocket(LISTENING_PORT);
            System.out.println("Listening on port " +
LISTENING_PORT);
            while (true) {
                connection = listener.accept();
                connectionQueue.add(connection);
            }
        }
        catch (Exception e) {
            System.out.println("Server shut down
unexpectedly.");
            System.out.println("Error:  " + e);
            return;
        }

    } // end main()


    /**
     * The class that defines the connection-handling
threads in the
     * thread pool.  The thread runs in an infinite loop in
which
     * it removes a connected socket from the connection
queue and
     * calls the handleConnection() method for that socket.
     */
    private static class ConnectionHandler extends Thread {
        File directory;  // The directory that contains the
files
                         // that are made available on this
server.
        ConnectionHandler(File directory) {
            this.directory = directory;
            setDaemon(true);
        }
        public void run() {
            while (true) {
                try {
                    Socket connection =
connectionQueue.take();
                    handleConnection(directory,connection);
                }
```

```
                catch (Exception e) {
                }
            }
        }
    }


    /**
     * This method processes process the connection with
one client.
     * It creates streams for communicating with the
client,
     * reads a command from the client, and carries out
that
     * command.  The connection is also logged to standard
output.
     * An output beginning with ERROR indicates that a
network
     * error occurred.  A line beginning with OK means that
     * there was no network error, but does not imply that
the
     * command from the client was a legal command.
     */
    private static void handleConnection(File directory,
Socket connection) {
        Scanner incoming;       // For reading data from
the client.
        PrintWriter outgoing;   // For transmitting data to
the client.
        String command = "Command not read";
        try {
            incoming = new Scanner(
connection.getInputStream() );
            outgoing = new PrintWriter(
connection.getOutputStream() );
            command = incoming.nextLine();
            if (command.equalsIgnoreCase("index")) {
                sendIndex(directory, outgoing);
            }
            else if
(command.toLowerCase().startsWith("get")){
                String fileName =
command.substring(3).trim();
                sendFile(fileName, directory, outgoing);
            }
            else {
                outgoing.println("unsupported command");
```

```
                outgoing.flush();
            }
            System.out.println("OK    " +
connection.getInetAddress()
                    + " " + command);
        }
        catch (Exception e) {
            System.out.println("ERROR " +
connection.getInetAddress()
                    + " " + command + " " + e);
        }
        finally {
            try {
                connection.close();
            }
            catch (IOException e) {
            }
        }
    }

    /**
     * This is called by the run() method in response to an
"index" command
     * from the client.  Send the list of files in the
server's directory.
     */
    private static void sendIndex(File directory,
PrintWriter outgoing) throws Exception {
        String[] fileList = directory.list();
        for (int i = 0; i < fileList.length; i++)
            outgoing.println(fileList[i]);
        outgoing.flush();
        outgoing.close();
        if (outgoing.checkError())
            throw new Exception("Error while transmitting
data.");
    }

    /**
     * This is called by the run() command in response to
"get <fileName>"
     * command from the client.  If the file doesn't exist,
send the message "error".
     * Otherwise, send the message "ok" followed by the
contents of the file.
     */
    private static void sendFile(String fileName, File
directory, PrintWriter outgoing) throws Exception {
```

```
        File file = new File(directory,fileName);
        if ( (! file.exists()) || file.isDirectory() ) {
            // (Note:  Don't try to send a directory, which
            // shouldn't be there anyway.)
            outgoing.println("error");
        }
        else {
            outgoing.println("ok");
            BufferedReader fileIn = new BufferedReader( new
FileReader(file) );
            while (true) {
                // Read and send lines from the file until
                // an end-of-file is encountered.
                String line = fileIn.readLine();
                if (line == null)
                    break;
                outgoing.println(line);
            }
        }
        outgoing.flush();
        outgoing.close();
        if (outgoing.checkError())
            throw new Exception("Error while transmitting
data.");
    }


} //end class ThreadedFileServer
```

Solution for Programming Exercise 12.6

Exercise 12.6:

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 π/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 π/4. If we choose a point in the square at random, the probability that it is inside the circle is π/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 π/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 point 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 *Labels* 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 12.3.5.)

Here is a picture of the program after it has run many 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);
Platform.runLater( () -> {
    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.)

Discussion

I will present three versions of the solution: one without a control button, one with the button and a single computation thread, and one that uses multiple threads. But first, you might be interested in how I set up the *Labels*:

```
countLabel =      new Label(" Number of Trials:   0");
piEstimateLabel = new Label(" Current Estimate:
(none)");
Label piLabel =  new Label(" Actual value of pi: " +
Math.PI + "  ");
String style = "-fx-font: bold 18pt monospaced; -fx-
padding: 8px; "
                  + "-fx-border-color: blue; -fx-border-
width:2px";
countLabel.setStyle(style);
piEstimateLabel.setStyle(style);
piLabel.setStyle(style);
countLabel.setMaxWidth(10000);
```

```
piEstimateLabel.setMaxWidth(10000);
piLabel.setMaxWidth(10000);

VBox root = new VBox(piLabel, piEstimateLabel,
countLabel);
```

Note the "style" that is applied to the labels. I used a monospaced font (in which all the characters are the same width) so that the characters on the three labels would line up neatly. I added some padding and a blue border to separate the labels more neatly. Then I set the maximum width of each label. That's done to allow the VBox to stretch the labels to fill the width of the window. Without that change, the size of the label would only be large enough to contain its text, and the border would not extend the full width of the window. Furthermore, the size of the label and the border would change when the text in the label is changed.

For the first version of the program, I simply wrapped the code given in the exercise for doing one batch of trials inside an infinite `while` loop, and made that the `run()` method of the thread. Note that the thread must use `Platform.runLater()` to run the code that modifies the GUI. (Such code must always be run on the application thread.)

```
private class ComputationThread extends Thread {
    final int BATCH_SIZE = 1000000;  // Number of trials
between updates of the display.
    long trialCount;     // Total number of trials that
have been performed.
    long inCircleCount;  // Number of trials in which
x*x+y*y is less than 1.
    public ComputationThread() {
        setDaemon(true);
        setPriority(Thread.currentThread().getPriority() -
1);
    }
    public void run() {
        while (true) {
            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);
            Platform.runLater( () -> {
                countLabel.setText(      " Number of
Trials:   " + trialCount);
                piEstimateLabel.setText( " Current
Estimate:   " + estimateForPi);
            } );

        }
    }
}
```

The thread is created and started in the application's `start()` method. It's quite possible that several batches of trials are done before the labels ever appear on the screen. That's OK because it's OK to call a label's `setText()` method even before the label appears.

The first version of the program is pretty short and simple, and it's not too hard to add the control button to get the second version. First of all, I added a `volatile` boolean variable, `running` to the program, and I added the following code at the beginning of the `while` loop in the thread to make the thread pause when the value of `running` is `false`:

```
synchronized(this) {
    while ( ! running ) { // wait for running to be true
        try {
            wait();
        }
         catch (InterruptedException e) {
        }
     }
 }
```

When `running` is `false`, this code calls the `wait()` method to go to sleep. It will require a call to `notify()` elsewhere in the program to wake the thread so that it can continue. Note that the synchronization is on the object "`this`", which refers here to the thread object. (Recall that `obj.wait()` must always be called in code that is synchronized on `obj`. In this case, `this` plays the role of `obj` because a simple call to `wait()` is really a call to

`this.wait()`.) A `while` loop is used in this code rather than an `if` statement to be absolutely sure that `running` is true before continuing.

When the thread is first started at the beginning of the program, the value of `running` is false. The thread sees this value and calls `wait()` as soon as it starts, and the thread remains in this waiting state until the user clicks the "Run" button. The method that responds to a button click has to set the value of `running` to `true`. It also has to call `notify()` to wake up the thread so it can see the new value. The `notify()` method must be called on the same object that the thread used to call `wait()`—that is, on the thread object itself. The program has a variable named `runner` that refers to the thread, so the thread is woken with the code:

```
synchronized(runner) {
    running = true;
    runner.notify();
}
```

It's important, by the way, that the statement that sets `running` to `true` is inside the `synchronized` statement here, and that the code in the thread that tests the value of `running` is also inside a statement that is synchronized on the same object. This avoids a race condition: Suppose these statements are not synchronized. Then it is possible that the thread might test the value of `running`, find it to be `false`, and decide to call `wait()`. However, before it actually calls `wait()`, the event-handling thread might set `running` to `true` **and** call `notify()`. So it's possible for the `notify()` to come before the `wait()`, which means the thread will be left waiting for a notification that was already given! Proper synchronization prevents the thread from being interrupted between the time it decides to call `wait()` and the time it actually does so.

As I mentioned in the statement of the exercise, I actually wanted to use several computation threads—one for each available processor. When I wrote that program, however, I was dismayed to find that it was much **slower** than the single-threaded program. After trying for a while to track down the reason, I eventually remembered that the call to `Math.random()` is synchronized internally (because there is a race condition in the way that the

next random number in a sequence is generated). There is a significant overhead involved in synchronization, which can really slow down a program when it is done too often. In this case, when `Math.random()` was called millions of times per second by several different threads, the slowdown was by a factor of about 100!

Behind the scenes, `Math.random()` uses an object of type `java.util.Random` to generate the random numbers. By having each thread create and use its own object of this type, instead of calling `Math.random()`, I avoided the synchronization problem. (I am honestly not sure how Java avoids the overhead of a `synchronized` statement when there is only one thread that is synchronizing on a given object.)

Aside from that problem, the main issue introduced by using multiple threads is how to combine the results from the different threads. In the first two versions of the program, the thread keeps track of the total number of trials and of the number of trials where `x*x+y*y` is less than 1. It uses this information to figure out what to put on the display labels. However, when there are two or more threads, the data that is needed to update the display is not available to any one thread. So, someone else has to collect the data from the threads and combine all the data to get the overall total number of trials and the overall total number of trials in which `x*x+y*y<1`.

There are several ways to do this. I decided to use a queue to transport the results from the computation threads to another thread that combines the results and updates the display accordingly. In fact, the role of the other thread is played by the JavaFX application thread. There is an *AnimationTimer* that handles the job. Its `handle()` method is called periodically. That method removes any data that has been collected in the queue and uses it to update the display. (One nice side effect of doing things this way is that `handle()` is called on the application thread, so it can update the display directly, without using `Platform.runLater()`.)

Trials are run in batches of 1,000,000. Each batch produces one result. The result is simply the number of trials in that batch for which `x*x+y*y<1`. The result is an integer, so for the result queue I use a queue of *Integers*. Each result in the queue corresponds to 1,000,000 trials, so each time I process a

result, I add 1000000 to the total number of trials and add the number from the queue to the number or trials in which `x*x+y*y<1`.

For the queue, I used a *LinkedBlockingQueue*. Since the queue is being used by several threads, I need a queue where the operations are properly synchronized. In fact, I don't need the blocking behavior of the blocking queue, but a *LinkedBlockingQueue* has a nice, synchronized method, `drainTo()`, for retrieving all the items from a queue in one step. I would have used *ConcurrentLinkedQueue* if it had a similar method.

I won't discuss the third version of the program further. You can see the source code, which is well commented, below.

The Solution

The first version of the program, with no control button:

```
import javafx.application.Application;
import javafx.application.Platform;
import javafx.stage.Stage;

import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.VBox;

/**
 * This program uses a probabilistic technique to estimate
the
 * value of the mathematical constant pi.  The technique is
to
 * choose random numbers x and y in the range 0 to 1, and
to
 * compute x*x + y*y.  The probability that x*x + y*y is
less than
 * 1 is pi/4.  If many trials are performed, and the number
of
 * trials in which x*x+y*y is less than 1 is divided by the
total
 * number of trials, the result is an approximation for
pi/4.
 * Multiplying this by 4 gives an approximation for pi.
 *
```

```
 * The program shows the estimate produced by this
procedure, along
 * with the number of trials that have been done and, for
comparison,
 * the actual value of pi.  These values are shown in three
Labels.
 * The computation is done by a separate thread that
updates the
 * contents of the labels after every millionth trial.
 *
 * In this version of the program, the computation thread
runs
 * continually from the time the program is started until
it
 * ends.  It is run at a reduced priority so that it does
not
 * interfere with the GUI thread.
 */
public class EstimatePi_1 extends Application {

    public static void main(String[] args) {
        launch();
    }
    //-------------------------------------------------------
----------------------

    private Label piEstimateLabel;  // A label for showing
the current estimate of pi.
    private Label countLabel;       // A label for showing
the number of trials.

    private ComputationThread runner;  // The thread that
does the computation.


    /**
     * Set up the GUI.  Create and start the computation
thread.
     */
    public void start(Stage stage) {

        countLabel =      new Label(" Number of Trials:
0");
        piEstimateLabel = new Label(" Current Estimate:
(none)");
        Label piLabel =  new Label(" Actual value of pi: "
+ Math.PI + "  ");
        String style = "-fx-font: bold 18pt monospaced; -
```

```
fx-padding: 8px; "
                            + "-fx-border-color: blue; -fx-
border-width:2px";
        countLabel.setStyle(style);
        piEstimateLabel.setStyle(style);
        piLabel.setStyle(style);
        countLabel.setMaxWidth(10000);
        piEstimateLabel.setMaxWidth(10000);
        piLabel.setMaxWidth(10000);

        VBox root = new VBox(piLabel, piEstimateLabel,
countLabel);
        root.setStyle("-fx-border-color: blue; -fx-border-
width:2px");
        stage.setScene(new Scene(root));
        stage.setTitle("Estimating Pi");
        stage.setResizable(false);
        stage.show();

        runner = new ComputationThread();
        runner.start();

    } // end start()


    /**
     *  This class defines the thread that does the
computation.
     *  The thread runs in an infinite loop in which it
performs
     *  batches of 1000000 trials and then updates the
display labels.
     */
    private class ComputationThread extends Thread {
        final int BATCH_SIZE = 1000000;  // Number of
trials between updates of the display.
        long trialCount;     // Total number of trials that
have been performed.
        long inCircleCount;  // Number of trials in which
x*x+y*y is less than 1.
        public ComputationThread() {
            setDaemon(true);

setPriority(Thread.currentThread().getPriority() - 1);
        }
        public void run() {
            while (true) {
                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);
                Platform.runLater( () -> {
                    countLabel.setText(      " Number of
Trials:   " + trialCount);
                    piEstimateLabel.setText( " Current
Estimate:   " + estimateForPi);
                });
            }
        }
    }


} // end class EstimatePi_1
```

The second version of the program, with a control button and just one thread. Significant changes from the first version are shown in red italic:

```
import javafx.application.Application;
import javafx.application.Platform;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.Button;
import javafx.scene.layout.VBox;
import javafx.scene.layout.BorderPane;
import javafx.geometry.Pos;
import javafx.geometry.Insets;

/**
 * This program uses a probabilistic technique to estimate
the
 * value of the mathematical constant pi.  The technique is
to
 * choose random numbers x and y in the range 0 to 1, and
to
 * compute x*x + y*y.  The probability that x*x + y*y is
less than
 * 1 is pi/4.  If many trials are performed, and the number
of
```

```
 * trials in which x*x+y*y is less than 1 is divided by the
total
 * number of trials, the result is an approximation for
pi/4.
 * Multiplying this by 4 gives an approximation for pi.
 *
 * The program shows the estimate produced by this
procedure, along
 * with the number of trials that have been done and, for
comparison,
 * the actual value of pi.  These values are shown in three
Labels.
 * The computation is done by a separate thread that
updates the
 * contents of the labels after every millionth trial.
 *
 * In this version of the program, there is a "Run"/"Pause"
button
 * that controls the computation thread.  Clicking the
button once
 * starts the thread; clicking it again pauses it.
Initially, the
 * thread is paused.
 */
public class EstimatePi_2 extends Application {

    public static void main(String[] args) {
        launch();
    }
    //-------------------------------------------------------
----------------------

    private Label piEstimateLabel;  // A label for showing
the current estimate of pi.
    private Label countLabel;       // A label for showing
the number of trials.

    private Button runPauseButton;  // Button to control
the thread.  Clicking this
                                    // button will pause
the thread if it is running
                                    // and will restart it
if it is paused.

    private ComputationThread runner;  // The thread that
does the computation.
```

```
    private volatile boolean running;   // Control variable
for signaling the thread to
                                        // run or pause.
Initially, this is false, so
                                        // the thread
pauses as soon as it is created,
                                        // until the user
clicks the "Run" button.


    /**
     * Set up GUI.  Create and start the computation thread
(but it immediately
     * goes to sleep to wait for running to be set to
true).
     */
    public void start(Stage stage) {

        countLabel =      new Label(" Number of Trials:
0");
        piEstimateLabel = new Label(" Current Estimate:
(none)");
        Label piLabel =  new Label(" Actual value of pi: "
+ Math.PI + "  ");
        String style = "-fx-font: bold 18pt monospaced; -
fx-padding: 8px; "
                + "-fx-border-color: blue; -fx-border-
width:2px";
        countLabel.setStyle(style);
        piEstimateLabel.setStyle(style);
        piLabel.setStyle(style);
        countLabel.setMaxWidth(10000);
        piEstimateLabel.setMaxWidth(10000);
        piLabel.setMaxWidth(10000);

        runPauseButton = new Button("Run");
        runPauseButton.setOnAction( e -> doRunPause() );

        VBox labels = new VBox(piLabel, piEstimateLabel,
countLabel);
        labels.setStyle("-fx-border-color: blue; -fx-
border-width:2px");

        BorderPane root = new BorderPane(labels);
        root.setBottom(runPauseButton);
        BorderPane.setAlignment(runPauseButton,
Pos.CENTER);
```

```
        BorderPane.setMargin(runPauseButton, new
Insets(10));


        stage.setScene(new Scene(root));
        stage.setTitle("Estimating Pi");
        stage.setResizable(false);
        stage.show();

        runner = new ComputationThread();
        runner.start();

    } // end start()


    /**
     * This method responds to clicks on the button, by
     * toggling the value of the signal variable from true
     * to false or from false to true.  The text on the
     * button is changed to match the state.  When
     * running is set to true, notify() is called to wake
     * up the thread.
     */
    public void doRunPause() {
        if (running) {
            runPauseButton.setText("Run");
            running = false;
        }
        else {
            runPauseButton.setText("Pause");
            synchronized(runner) {
                running = true;
                runner.notify();
            }
        }
    }


    /**
     *  This class defines the thread that does the
computation.
     *  The thread runs in an infinite loop in which it
performs
     *  batches of 1000000 trials and then updates the
display labels.
     */
```

```
    private class ComputationThread extends Thread {
        final int BATCH_SIZE = 1000000;  // Number of
trials between updates of the display.
        long trialCount;     // Total number of trials that
have been performed.
        long inCircleCount;  // Number of trials in which
x*x+y*y is less than 1.
        public ComputationThread() {
            setDaemon(true);

setPriority(Thread.currentThread().getPriority() - 1);
        }
        public void run() {
            while (true) {
                synchronized(this) {
                    while ( ! running ) { // wait for
running to be true
                        try {
                            wait();
                        }
                        catch (InterruptedException e) {
                        }

                    }
                }
                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);
                Platform.runLater( () -> {
                    countLabel.setText(      " Number of
Trials:   " + trialCount);
                    piEstimateLabel.setText( " Current
Estimate:   " + estimateForPi);
                });
            }
        }
    }


} // end class EstimatePi_2
```

The third version of the program, with a control button and multiple threads. Significant changes from the previous versions are shown in red italic

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.Button;
import javafx.scene.layout.VBox;
import javafx.scene.layout.BorderPane;
import javafx.geometry.Pos;
import javafx.geometry.Insets;
import javafx.animation.AnimationTimer;

import java.util.concurrent.LinkedBlockingQueue;
import java.util.ArrayList;
import java.util.Random;

/**
 * This program uses a probabilistic technique to estimate
the
 * value of the mathematical constant pi.  The technique is
to
 * choose random numbers x and y in the range 0 to 1, and
to
 * compute x*x + y*y.  The probability that x*x + y*y is
less than
 * 1 is pi/4.  If many trials are performed, and the number
of
 * trials in which x*x+y*y is less than 1 is divided by the
total
 * number of trials, the result is an approximation for
pi/4.
 * Multiplying this by 4 gives an approximation for pi.
 *
 * The program shows the estimate produced by this

procedure, along
 * with the number of trials that have been done and, for
comparison,
 * the actual value of pi.  These values are shown in three
Labels.
 * The computation is done by several separate threads.
```

```
         -                    -            -
Periodically,
 * these threads place their results into a queue.  Another
thread,
 * the event-handling thread in a method called by a Timer,
removes
 * results from the queue and applies them to the labels
every
 * 1/60 second.
 *
 * In this version of the program, the computation thread
runs
 * continually from the time the program is started until
it
 * ends.  It is run at a reduced priority so that it does
not
 * interfere with the GUI thread.
 */
public class EstimatePi_3 extends Application {

    public static void main(String[] args) {
        launch();
    }
    //-------------------------------------------------------
----------------------

    private final static int BATCH_SIZE = 1000000;
                                // This is the number of
trials
                                // in a batch.  A
computation thread runs this
                                // many trials in a fast
for loop, without checking
                                // the value of running and
without reporting its
                                // results.  After the for
loop, the thread puts
                                // the result from that
batch into the queue.  It
                                // then checks the value of
running and will pause
                                // if running is false.

    private long totalTrialCount;    // Total number of
trials considered so far.
    private long totalInCircleCount; // Number of those
trials for which x*x+y*y < 1.
```

```
    private Label piEstimateLabel;  // A label for showing
the current estimate of pi.
    private Label countLabel;       // A label for showing
the number of trials.

    private Button runPauseButton;  // Button to control
the threads.  Clicking this
                                    // button will pause
the threads if they are
                                    // running and will
restart it if it is paused.

    private volatile boolean running;   // Control variable
for signaling the threads
                                        // to run or pause.
Initially, this is false,
                                        // so the thread
pauses as soon as it is
                                        // created, until
the user clicks "Run."


    private LinkedBlockingQueue<Integer> resultsQueue;
                                    // Results from the
computation
                                    // threads are placed into
this queue.  Every number
                                    // in the queue represents
the results from running
                                    // a batch of trials, of
size BATCH_SIZE.  The number
                                    // in the queue is the
number of trials in that batch
                                    // that resulted in
x*x+y*y being less than 1.  (Note
                                    // that I use a blocking
queue rather than a
                                    // ConcurrentLinkedQueue
only because the blocking
                                    // queue has a convenient

drainTo() method for getting
                                    // all the items out of
the queue at once, with correct
                                    // synchronization.)

    private AnimationTimer resultsTimer;
```

```
    private AnimationTimer resultsTimer;
                                    // While the computation is
running, this timer is
                                    // also running.  Every
1/60 second, it grabs the
                                    // results from the queue
and applies them to the
                                    // display labels.  (Note
that some results can be
                                    // left in the queue while
the timer and threads are
                                    // paused.  This seems
harmless.)


    /**
     * Sets up the GUI.  Creates computation threads and
starts them.
     * Creates the AnimationTimer but does not start it.
     */
    public void start(Stage stage) {

        countLabel =      new Label(" Number of Trials:
0");
        piEstimateLabel = new Label(" Current Estimate:
(none)");
        Label piLabel =  new Label(" Actual value of pi: "
+ Math.PI + "  ");
        String style = "-fx-font: bold 18pt monospaced; -
fx-padding: 8px; "
                + "-fx-border-color: blue; -fx-border-
width:2px";
        countLabel.setStyle(style);
        piEstimateLabel.setStyle(style);
        piLabel.setStyle(style);
        countLabel.setMaxWidth(10000);
        piEstimateLabel.setMaxWidth(10000);
        piLabel.setMaxWidth(10000);

        runPauseButton = new Button("Run");
        runPauseButton.setOnAction( e -> doRunPause() );

        VBox labels = new VBox(piLabel, piEstimateLabel,
countLabel);
        labels.setStyle("-fx-border-color: blue; -fx-
border-width:2px");

        BorderPane root = new BorderPane(labels);
```

```
        root.setBottom(runPauseButton);
        BorderPane.setAlignment(runPauseButton,
Pos.CENTER);
        BorderPane.setMargin(runPauseButton, new
Insets(10));


        stage.setScene(new Scene(root));
        stage.setTitle("Estimating Pi");
        stage.setResizable(false);
        stage.show();

        resultsTimer = new AnimationTimer() {
                        // (Must be created before threads,
since
                        //    the threads use it for
synchronization.)
            public void handle(long time) {
                grabResults();
            }
        };

        resultsQueue = new LinkedBlockingQueue<Integer>();
        int threadCount =
Runtime.getRuntime().availableProcessors();
        for (int i = 0; i < threadCount; i++) {
            ComputationThread runner = new
ComputationThread();
            runner.start();
        }

    } // end start


    /**
     * This method responds to clicks on the button, by
     * toggling the value of the signal variable from true
     * to false or from false to true.  The text on the
     * button is changed to match the state. The timer
     * is also stopped or started.
     */
    public void doRunPause() {
        if (running) {
            resultsTimer.stop();
            runPauseButton.setText("Run");
            running = false;
        }
```

```
        else {
            runPauseButton.setText("Pause");
            resultsTimer.start();
            synchronized(resultsTimer) {
                    // IMPORTANT: Synchronization is now on
ApplicationTimer!
                running = true;
                resultsTimer.notifyAll();
                    // IMPORTANT: Use notifyAll(), not
notify(),
                    // to wake ALL computation threads
            }
        }
    }


    /**
     * This method is called by the timer, every 1/60
second while the
     * computation is running.  It grabs the entire
contents of the
     * queue that is used to send results from the threads
to
     * this method.  Each value in the queue represents the
number of
     * trials, out of a batch of size BATCH_SIZE, in which
x*x+y*Y was
     * less than 1.  This method updates the total number
of trials that
     * have been performed and the total number of trials
for which
     * x*x+y*y was less than 1.  It then updates the
display labels
     * with the new data.
     */
    private void grabResults() {
        ArrayList<Integer> results = new ArrayList<Integer>
();
        resultsQueue.drainTo(results);  // Get entire
contents of queue.
                    // Using this method avoids having to
synchronize
                    // the entire process of removing items
from the
                    // queue one at a time.  (And doing
that without
```

```
that without
                    // synchronization would introduce a
race condition.)
        for (int inCircleCount : results) {
            totalTrialCount += BATCH_SIZE;
            totalInCircleCount += inCircleCount;
        }
        double estimateOfPi = 4 *
((double)totalInCircleCount / totalTrialCount);
        countLabel.setText(      " Number of Trials:   " +
totalTrialCount);
        piEstimateLabel.setText( " Current Estimate:   " +
estimateOfPi);
        // System.out.println("Got " + results.size() + "
results.");  // for testing
    }


    /**
     *  This class defines the threads that do the
computation.
     *  Each thread runs in an infinite loop in which it
performs
     *  batches of 1000000 trials and places the result in
the queue.
     *  Just after it starts and between batches, the
thread tests
     *  the value of the signal variable, running.  If this
variable
     *  is false, then the thread sleeps until the value of
running
     *  is set to true.  Note that this method creates and
uses its
     *  own object of type Random to generate random
numbers. (Because
     *  access to Math.random() has to be synchronized,
using it
     *  in multiple threads slowed things down immensely.)
     *  Synchronization in this thread, as in the rest of
the program,
     *  is on the ApplicationTimer object, which is
referred to here as
     *  "EstimatePi_3.this".  The previous version used the
thread
     *  object for synchronization, but in this version
there can
     *  be multiple threads, so it seemed more natural to
```

```
use something else
     */
    private class ComputationThread extends Thread {
        public ComputationThread() {
            setDaemon(true);

setPriority(Thread.currentThread().getPriority() - 1);
        }
        public void run() {
            Random myRandom = new Random();
            while (true) {
                synchronized(resultsTimer) {
                    while ( ! running ) {

                        try {
                            resultsTimer.wait();
                        }
                        catch (InterruptedException e) {
                        }
                    }
                }
                int inCircleCount = 0;
                for (int i = 0; i < BATCH_SIZE; i++) {
                    double x = myRandom.nextDouble();
                    double y = myRandom.nextDouble();
                    if (x*x + y*y < 1)
                        inCircleCount++;
                }
                resultsQueue.add(inCircleCount);
            }
        }
    }

} // end class EstimatePi_3
```

Solution for Programming Exercise 12.7

Exercise 12.7:

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 established. It should throw an *IOException* if some error occurs during the setup process. Note that any messages that are sent by the hub as part of the handshake must 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. (Just imitate the code that asks for the host name.) You will have to decide what to do if two users want to use the same name.

For the second improvement, personal messages, I suggest writing a new *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 of 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 by the hub 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. Don't forget that *PrivateMessage* needs to be declared to implement *Serializable*.

If you attempt this exercise, you are likely to find it quite challenging.

Discussion

My solution defines a subclass, *NewChatRoomHub*, of *Hub* to make it possible to manage user names. I also wrote *NewChatRoomWindow* class, but not as a subclass; instead it is a modified copy of *ChatRoomWindow*.

In the *Hub* class, users are identified by ID numbers. In some ways, it would have been easier to modify *Hub*, instead of creating a subclass. However, I wanted to build on the package netgame.common, without modifying the existing classes. The *NewChatRoomHub* class has an instance variable

```
private TreeMap<Integer,String> nameMap = new
TreeMap<Integer,String>();
```

to keep track of client names by associating each client's name to that client's ID number. That is, `nameMap.get(id)` is the name of the client whose ID number is `id`. There has to be a way to add a user name to `nameMap` when a new client connects, and there has to be a way to remove a client from

`nameMap` when that client disconnects. There also has to be a way for a client to get a name in the first place.

It might have been nice to have a list of authorized users, and give each authorized user a name and a password. The hub would then require a name and a password before it would let a client connect. The list of names and passwords could be stored in a file or, if there are a lot of them, in a database. However, I decided to keep things simple. Clients choose their names when they connect. The client's selected name is sent to the hub as part of the `extraHandshake()` method. To avoid having duplicate names, the hub will modify the client's name if there is already someone in the chat room using that name. For example, if the client wants to be "Fred" and there is already a Fred, then the hub will assign the name "Fred#2" to the client. Since the hub can modify the name, it sends the client's name back to the client, possibly with some modification. The client reads the returned name as part of the handshake. To implement this idea, the `extraHandshake()` method in the client simply sends the client's requested name to the hub by writing the name to the connection's output stream. It then reads the possibly modified name from the connection's input stream:

```
protected void extraHandshake(ObjectInputStream in,
ObjectOutputStream out)

throws IOException {
    try {
        out.writeObject(myName);  // Send user's name
request to the server.
        myName = (String)in.readObject();  // Get the
possibly modified name.
    }
    catch (Exception e) {
        throw new IOException("Error while setting up
connection: " + e);
    }
}
```

(This code is from the *ChatClient* class, which is a nested subclass of *NewChatRoomWindow*.)

The corresponding `extraHandshake()` method in the *NewChatRoomHub* class has to read the name that was sent by the client, possibly modify the

name, and write the name back to the client. It also adds the client to `nameMap`, which contains the names of all connected clients. My code is very careful to produce a name that is non-null, non-empty, and unique:

```
protected void extraHandshake(int playerID,
                  ObjectInputStream in, ObjectOutputStream
out) throws IOException {
    try {
        String name = (String)in.readObject(); // Read
requested name from client.
        if (name == null)
            name = "noname";
        if (name.length() > 15)
            name = name.substring(0,15).trim();
        if (name.equals(""))
            name = "noname";
        synchronized(nameMap) {
               // Synchronized to be absolutely sure that
there will be
               // no duplicate names.
            if (nameMap.containsValue(name)) {
                String approvedName = name;
                int num = 2;
                while
(nameMap.containsValue(approvedName)) {
                    approvedName = name + "#" + num;
                    num++;
                }
                name = approvedName;
            }
        }
        out.writeObject(name);      // Send actual name to
client.
        nameMap.put(playerID,name); // Add client's name
to nameMap.
    }
    catch (Exception e) {
        throw new IOException("Error while setting up
connection: " + e);
    }
}
```

In order to get private messages from one player to another, I defined a *PrivateMessage* class, as suggested in the exercise. An object `pm` of type *PrivateMessage* has public instance variables `pm.senderID` and

`pm.recipientID` to hold the ID numbers of the sender and the recipient, and it has `pm.message` to hold the message itself. The hub has to know what to do with such a message. Messages are processed in the hub by the `messageReceived()` method. I overrode this method in *NewChatRoomHub* to handle *PrivateMessage* and to pass messages of other types on to the superclass. One interesting point is that the hub ignores the `senderID` in the incoming message and sets the `senderID` in the outgoing message to be the ID number of the client that actually sent the message. This is done to prevent clients from sending forged messages that appear to be from someone else. Here is the `messageReceived()` method from *NewChatRoomHub*:

```
/**
 * This method is overridden to provide support for
PrivateMessages.
 * If a PrivateMessage is received from some client, this
method
 * will set the senderID field in the message to be the ID
number
 * of the client who sent the message.  It will then send
the
 * message on to the specified recipient.  If some other
type
 * of message is received, it is handled by the
messageReceived()
 * method in the superclass (which will wrap it in a
ForwardedMessage
 * and send it to all connected clients).
 */
protected void messageReceived(int playerID, Object
message) {
    if (message instanceof PrivateMessage) {
        PrivateMessage pm = (PrivateMessage)message;
        pm.senderID = playerID;
        sendToOne(pm.recipientID, pm);
    }
    else
        super.messageReceived(playerID, message);
}
```

It is easy enough to add some extra inputs to the original window from *ChatRoomWindow* to make it possible for the user to send private messages. Just add a text input box for the message, a button to send the message, and a *ComboBox* that contains a list of possible recipients. (The ComboBox

control has not yet been covered in this book, but see Subsection 13.3.3.) The user enters a message, selects the recipient from the list, and clicks the button to send. The program constructs a *PrivateMessage* containing the message and the ID number of the recipient, and it sends the *PrivateMessage* to the hub, which forwards it on to the intended recipient. This code can be found in *NewChatRoomWindow.*

At least, it **would** be easy if not for the fact that I want users of the chat room to be identified by name, not ID number. The whole point of using names is that they provide a more meaningful way to refer to users. Each client already gets a list of ID numbers of all connected clients, but I want each client to have a list of **names** of connected clients. The hub has that information. I just needed a way to get that information to the clients. Since the client list changes every time a client connects or disconnects, the hub will have to send new information to the clients every time that happens. I was able to program this behavior by overriding the `playerConnected()` and `playerDisconnected()` methods from the *Hub* class. My *NewChatRoomHub* class includes the following definitions for those methods (which do nothing in class *Hub*):

```
/**
 *  This method is called when a new client connects.  It
is called
 *  after the extraHandshake() method has been called, so
that the
 *  client's name has already been added to nameMap.  This
method
 *  creates a ClientConnectedMessage and sends it to all
connected
 *  clients to announce the new participant in the chat
room.
 */
protected void playerConnected(int playerID) {
    resetOutput(); // Reset the output stream before
resending nameMap.
    sendToAll(new
ClientConnectedMessage(playerID,nameMap));
}

/**
 * This method is called when a client has been
disconnected from
```

```
 * this hub.  It removes the client from the nameMap and
sends
 * a ClientDisconnectMessage to all connected players to
 * announce the fact that the client has left the chat
room.
 */
protected void playerDisconnected(int playerID) {
    String name = nameMap.get(playerID); // Get the
departing player's name.
    nameMap.remove(playerID);  // Remove the player from
nameMap.
    resetOutput(); // Reset the output stream before
resending nameMap.
    sendToAll(new ClientDisconnectedMessage(playerID,
name, nameMap));
}
```

It actually took me quite a while to get this right. To simplify the processing, I ended up defining new message types, *ClientConnectedMessage* and *ClientDisconnectedMessage*, to hold the information that I wanted to send to the clients. A *ClientConnectedMessage* includes the ID number of the client who has just connected, as well as the `nameMap` that contains the ID numbers and names of all connected clients. When a client receives this message, it can inform the user that someone has entered the chat room, and it can use the `nameMap` to construct the list of names of possible recipients of private messages. *ClientDisconnectedMessages* are similar.

Not for the first time, I introduced a bug by forgetting that an *ObjectOuputStream* has to be reset if it is used to send the same object twice, with modifications between the two transmissions. This is necessary in this example because the same object, `nameMap`, is transmitted as part of every *ClientConnectedMessage* and every *ClientDisconnectedMessage*. When I didn't reset the output, the changes that I made to that object were not seen by the clients that received those messages. In my code, the method `resetOuput()` will cause the output stream leading to every client to be reset.

Back in the *NewChatRoomWindow* class, the chat room client has to be prepared to receive *PrivateMessages*, *ClientConnectedMessages*, and *ClientDisconnectedMessages*. It can also receive *ForwardedMessages*

which are used, as in the original chat room application, to broadcast a message from one client to all connected clients. In each case, the client will add an appropriate message to the transcript. It uses `clientNameMap`—the local copy of `nameMap`—to translate ID numbers from the messages into the names of the corresponding chat room users:

```
protected void messageReceived(Object message) {
    if (message instanceof ForwardedMessage) {
        ForwardedMessage fm = (ForwardedMessage)message;
        String senderName =
clientNameMap.get(fm.senderID);
        addToTranscript(senderName + " SAYS:  " +
fm.message);
    }
    else if (message instanceof PrivateMessage) {
        PrivateMessage pm = (PrivateMessage)message;
        String senderName =
clientNameMap.get(pm.senderID);
        addToTranscript("PRIVATE MESSAGE FROM " +
senderName + ":  " + pm.message);
    }
    else if (message instanceof ClientConnectedMessage) {
        ClientConnectedMessage cm =
(ClientConnectedMessage)message;
        addToTranscript('"' +
cm.nameMap.get(cm.newClientID) + "\" HAS JOINED THE CHAT
ROOM.");
        newNameMap(cm.nameMap);
    }
    else if (message instanceof ClientDisconnectedMessage)
{
        ClientDisconnectedMessage dm =
(ClientDisconnectedMessage)message;
        addToTranscript('"' +
clientNameMap.get(dm.departingClientID) + "\" HAS LEFT THE
CHAT ROOM.");
        newNameMap(dm.nameMap);
    }
}
```

The `newNameMap()` method, which is called twice in the above code, will assign the `nameMap` from the message to `clientNameMap`. It also updates the *ComboBox*, `clientList`, which stores the list of names of connected users (leaving out the name of the user who is using the program, since he won't

want to send a private message to himself). If the user who is currently selected is still in the new list, then that user is selected again.

```
private void newNameMap(final TreeMap<Integer,String>
nameMap) {
    Platform.runLater( () ->  {
        clientNameMap = nameMap;
        String currentlySelected =
clientList.getSelectionModel().getSelectedItem();
        clientList.getItems().clear();
        boolean someoneIsThere = false;
        boolean currentSelectionIsThere = false;
        for (String str: nameMap.values()) {
            if (!str.equals(myName)) {
                clientList.getItems().add(str);
                someoneIsThere = true;
            }
            if (str.equals(currentlySelected))
                currentSelectionIsThere = true;
        }
        privateMessageInput.setEditable(someoneIsThere);
        privateMessageInput.setDisable(!someoneIsThere);
        sendPrivateButton.setDisable(!someoneIsThere);
        if (!someoneIsThere)
            clientList.getItems().add("(no one
available)");
        if (currentSelectionIsThere)

clientList.getSelectionModel().select(currentlySelected);
        else
            clientList.getSelectionModel().select(0);
    });
}
```

The complete source code for the new chat room application is shown below. It consists of six classes, which I defined in a package named `netgame.newchat`. These classes depend on the classes from the netgame.common package.

The Solution

`NewChatRoomServer` is a short class with a main routine that simply creates a hub that acts as the server for the application:

```
package netgame.newchat;

import java.io.IOException;

/**
 * This class contains just a small main class that creates
a NewChatRoomHub
 * and starts it listening on port 37830.  This port is
used
 * by the NewChatRoomWindow application.  This program
should be run
 * on the computer that "hosts" the chat room.  See the
NewChatRoomWindow
 * class for more details.  Once the server starts
listening, it
 * will listens for connection requests from clients until
the
 * NewChatRoomServer program is terminated (for example by
a
 * Control-C).
 */
public class NewChatRoomServer {

    private final static int PORT = 37830;

    public static void main(String[] args) {
        try {
            new NewChatRoomHub(PORT);
        }
        catch (IOException e) {
            System.out.println("Can't create listening
socket.  Shutting down.");
        }
    }

}
```

There are three short classes that define various types of messages which are used in the application:

```
package netgame.newchat;

import java.io.Serializable;
```

```
/**
 * Represents a string sent as a message from one client
 * to another client.  Note:  The ChatRoomHub will set the
 * senderID of a PrivateMessage to be the ID number of the
 * client who actually sent the message, in order to avoid
 * the possibility of rogue clients that try to forge
 * messages that appear to come from other clients.
 */
public class PrivateMessage implements Serializable {

    public int senderID;    // The ID number of the sender.
    public int recipientID; // The ID number of the
recipient.
    public String message;  // The message.

    /**
     *  Create a private message from one user to another.
     *  The senderID of the message will be set by the hub.
     */
    public PrivateMessage(int recipientID, String message)
{
        this.recipientID = recipientID;
        this.message = message;
    }

}



package netgame.newchat;

import java.io.Serializable;
import java.util.TreeMap;

/**
 * A message of this type will be sent by the hub to all
 * connected clients when a new client joins the chat room
 */
public class ClientConnectedMessage implements Serializable
{

    public int newClientID;  // The ID number of the client
who has connected.
    public TreeMap<Integer,String> nameMap;  // Map of all
connected client IDs to their names.

    public ClientConnectedMessage(int newClientID,
TreeMap<Integer,String> nameMap) {
```

```
        this.newClientID = newClientID;
        this.nameMap = nameMap;
    }

}



package netgame.newchat;

import java.io.Serializable;
import java.util.TreeMap;

/**
 * A message of this type will be sent by the hub to all
 * remaining connected clients when a client leaves the
 * chat room.
 */
public class ClientDisconnectedMessage implements
Serializable {

    public int departingClientID;  // The ID number of the
client who has left the chat room.
    public String departingClientName;  // The name of the
departing client
    public TreeMap<Integer,String> nameMap;  // Map of all
connected client IDs to their names.
                                             //  (Note that
the departing client is not included.)

    public ClientDisconnectedMessage(int departingClientID,
            String departingClientName,
TreeMap<Integer,String> nameMap) {
        this.departingClientID = departingClientID;
        this.departingClientName = departingClientName;
        this.nameMap = nameMap;
    }


}
```

The subclass of *Hub* that defines the server for the application:

```
package netgame.newchat;
```

```
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.TreeMap;

import netgame.common.*;

/**
 *  This class defines the "hub" that acts as a server for
the
 *  chat room application.  It extends the basic Hub class
in
 *  order to support names for clients, as well as ID
numbers.
 */
public class NewChatRoomHub extends Hub {

    /**
     * This map keeps track of the names of all connected
clients.
     * It maps client ID numbers to client names.
     */
    private TreeMap<Integer,String> nameMap = new
TreeMap<Integer,String>();

    /**
     * Create a NewChatRoomHub, which will listen for
connections on
     * a specified port.
     * @param port the port on which to listen for
connections
     * @throws IOException if it is not possible to create
a listening socket
     */
    public NewChatRoomHub(int port) throws IOException {
        super(port);
    }

    /**
     * This method is called as part of the connection
setup between this hub
     * and a client that has requested a connection.  It is
overridden in this
     * class so that a name can be assigned to the client
as part of the setup
     * process.  This method works in cooperation with the
extraHandshake()
     * method in the client class (which is defined as a
```

```
nested class inside
     * NewChatRoomWindow).  In this method, the Hub reads a
string from the
     * user that contains the name that the client wants to
use.  The name
     * can be modified to make sure that it is non-null, 15
characters or less.
     * The resulting name is further modified by adding a
suffix such as
     * "#2" or "#3" if the name is already in use by
another client.  Finally,
     * the possibly modified name is sent back to the
client, which will use
     * the returned value as the name that identifies the
client in the chat
     * room.
     */
    protected void extraHandshake(int playerID,
                      ObjectInputStream in,
ObjectOutputStream out) throws IOException {
        try {
            String name = (String)in.readObject();
            if (name == null)
                name = "noname";
            if (name.length() > 15)
                name = name.substring(0,15).trim();
            if (name.equals(""))
                name = "noname";
            synchronized(nameMap) {
                if (nameMap.containsValue(name)) {
                    String approvedName = name;
                    int num = 2;
                    while
(nameMap.containsValue(approvedName)) {
                        approvedName = name + "#" + num;
                        num++;
                    }
                    name = approvedName;
                }
            }
            out.writeObject(name);
            nameMap.put(playerID,name);
        }
        catch (Exception e) {
            throw new IOException("Error while setting up
connection: " + e);
        }
    }
```

```
    /**
     * This method is overridden to provide support for
PrivateMessages.
     * If a PrivateMessage is received from some client,
this method
     * will set the senderID field in the message to be the
ID number
     * of the client who sent the message.  It will then
send the
     * message on to the specified recipient.  If some
other type
     * of message is received, it is handled by the
messageReceived()
     * method in the superclass (which will wrap it in a
ForwardedMessage
     * and send it to all connected clients).
     */
    protected void messageReceived(int playerID, Object
message) {
        if (message instanceof PrivateMessage) {
            PrivateMessage pm = (PrivateMessage)message;
            pm.senderID = playerID;
            sendToOne(pm.recipientID, pm);
        }
        else
            super.messageReceived(playerID, message);
    }

    /**
     *  This method is called when a new client connects.
It is called
     *  after the extraHandshake() method has been called,
so that the
     *  client's name has already been added to nameMap.
This method
     *  creates a ClientConnectedMessage and sends it to
all connected
     *  clients to announce the new participant in the chat
room.
     */
    protected void playerConnected(int playerID) {
        resetOutput(); // Reset the output stream before
resending nameMap.
        sendToAll(new
ClientConnectedMessage(playerID,nameMap));
    }
```

```
    /**
     * This method is called when a client has been
disconnected from
     * this hub.  It removes the client from the nameMap
and sends
     * a ClientDisconnectedMessage to all connected players
to
     * announce the fact that the client has left the chat
room.
     */
    protected void playerDisconnected(int playerID) {
        String name = nameMap.get(playerID); // Get the
departing player's name.
        nameMap.remove(playerID);  // Remove the player
from nameMap.
        resetOutput(); // Reset the output stream before
resending nameMap.
        sendToAll(new ClientDisconnectedMessage(playerID,
name, nameMap));
    }

}
```

And finally, the class that defines the clients, *NewChatRoomWindow*. This class is a modified version of ChatRoomWindow from the original chat room application. Significant changes from the original are shown in red italic.

```
package netgame.newchat;

import javafx.application.Application;
import javafx.application.Platform;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.ComboBox;
import javafx.scene.control.TextField;
import javafx.scene.control.TextArea;
import javafx.scene.control.Alert;
import javafx.scene.control.TextInputDialog;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;
```

```
import javafx.scene.layout.VBox;
import javafx.scene.layout.Priority;
import javafx.geometry.Insets;

import java.util.Optional;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.TreeMap;

import netgame.common.*;

/**
 * This class represents a client for a "chat room"
application.  The chat
 * room is hosted by a server running on some computer.
The user of this
 * program must know the host name (or IP address) of the
computer that
 * hosts the chat room.  When this program is run, it asks
for that
 * information and for the name that the user wants to use
in the chat
 * room.  Then, it opens a window that has an input box
where the
 * user can enter messages to be sent to the chat room.
The message is
 * sent when the user presses return in the input box or
when the
 * user clicks a Send button.  There is also a text area
that shows
 * a transcript of all messages from participants in the
chat room.
 * <p>The user can also send private messages to individual
users.
 * The user selects the recipient's name from a pop-up list
of
 * connected users.
 * <p>Participants in the chat room are represented by ID
numbers
 * that are assigned to them by the server when they
connect. They
 * also have names which they select.
 */
public class NewChatRoomWindow extends Application {
```

```
    public static void main(String[] args) {
        launch(args);
    }
    //-------------------------------------------------------
----------------------------

    private final static int PORT = 37830; // The ChatRoom
port number; can't be
                                           // changed here
unless the ChatRoomServer
                                           // program is
also changed.

    private TextField messageInput;   // For entering
messages to be sent to the chat room
    private Button sendButton;        // Sends the contents
of the messageInput.
    private Button quitButton;        // Leaves the chat
room cleanly, by sending a DisconnectMessage

    private TextArea transcript;      // Contains all
messages sent by chat room participant, as well
                                      //    as a few
additional status messages,
                                      //    such as when a
new user arrives.

    private ChatClient connection;    // Represents the
connection to the Hub; used to send messages;
                                      // also receives and
processes messages from the Hub.

    private volatile boolean connected; // This is true
while the client is connected to the hub.

    private volatile String myName; // The name that this
client uses in the chat room.
                                    // Originally selected
by the user, but might be modified
                                    // if there is already
a client of the same name connected
                                    // to the Hub.

    private volatile TreeMap<Integer,String> clientNameMap
= new TreeMap<Integer, String>();
                                    // The clientNameMap
maps client ID numbers to the names that they are
```

```
                                        // using in the chat
room.  Every time a client connects or disconnects,
                                        // the Hub sends a new,
modified name map to each connected client.  When
                                        // that message is
received, the clientNameMap is replaced with the new value,
                                        // and the content of
the clientList is replaced with info from the nameMap.

    private ComboBox<String> clientList;    // List of
connected client names, where the user can select
                                                //   the client
who is to receive the private message.

    private TextField privateMessageInput;  // For entering
messages to be set to individual clients.
    private Button sendPrivateButton;   // Sends the
contents of privateMesssageInput to the user selected
                                            //   in the
clientList.

    /**
     * Gets the host name (or IP address) of the chat room
server from the
     * user and then opens the main window.  The program
ends when the user
     * closes the window.
     */
    public void start( Stage stage ) {

        TextInputDialog question = new TextInputDialog();
        question.setHeaderText("Enter the host name of
the\ncomputer that hosts the chat room.");
        question.setContentText("Host Name:");
        Optional<String> response = question.showAndWait();
        if ( ! response.isPresent() )
            System.exit(0);
        String host = response.get().trim();
        if (host == null || host.trim().length() == 0)
            System.exit(0);

        question = new TextInputDialog();
        question.setHeaderText("Enter the name that you
want\nto use in the chat room.");
        question.setContentText("Your Name:");
        response = question.showAndWait();
```

```
        if ( ! response.isPresent() )
            System.exit(0);
        myName = response.get().trim();
        if (myName == null || myName.trim().length() == 0)
            System.exit(0);

        transcript = new TextArea();
        transcript.setPrefRowCount(30);
        transcript.setPrefColumnCount(60);
        transcript.setWrapText(true);
        transcript.setEditable(false);

        sendButton = new Button("send to all");
        quitButton = new Button("quit");
        messageInput = new TextField();
        messageInput.setPrefColumnCount(40);
        sendButton.setOnAction( e -> doSend() );
        quitButton.setOnAction( e -> doQuit() );
        sendButton.setDisable(true);
        messageInput.setEditable(false);
        messageInput.setDisable(true);

        sendPrivateButton = new Button("send to one");
        sendPrivateButton.setOnAction( e ->
doSendPrivateMessage() );
        privateMessageInput = new TextField();
        privateMessageInput.setPrefColumnCount(30);
        clientList = new ComboBox<String>();
        clientList.setEditable(false);
        clientList.getItems().add("(no one available)");
        clientList.getSelectionModel().select(0);

        HBox bottomRow1 = new HBox(8, new Label("YOU
SAY:"), messageInput, sendButton, quitButton);
        HBox.setHgrow(messageInput, Priority.ALWAYS);
        HBox.setMargin(quitButton, new Insets(0,0,0,50));

        HBox bottomRow2 = new HBox(8, new Label("SAY:"),
privateMessageInput,
                                        new Label(" To: "),
clientList, sendPrivateButton);
        HBox.setHgrow(privateMessageInput,
Priority.ALWAYS);

        VBox bottom = new VBox(8, bottomRow1, bottomRow2);
        bottom.setPadding(new Insets(8));
```

```
        bottom.setStyle("-fx-border-color: black; -fx-
border-width:2px");
        BorderPane root = new BorderPane(transcript);
        root.setBottom(bottom);

        stage.setScene( new Scene(root) );
        stage.setTitle("Networked Chat");
        stage.setResizable(false);
        stage.setOnHidden( e -> doQuit() );
        stage.show();

        /* The next two lines make the sendButton and
sendPrivateButton into the
         * default button for the window exactly when the
corresponding input box
         * is focussed.  This means that the user can just
hit return while
         * typing in an input box to send the message. */

        messageInput.focusedProperty().addListener(
                        (target,oldVal,newVal) ->
sendButton.setDefaultButton(newVal) );
        privateMessageInput.focusedProperty().addListener(
                         (target,oldVal,newVal) ->
sendPrivateButton.setDefaultButton(newVal) );

        new Thread() {
                // This is a thread that opens the
connection to the server.  Since
                // that operation can block, it's not done
directly in the constructor.
                // Once the connection is established, the
user interface elements are
                // enabled so the user can send messages.
The Thread dies after
                // the connection is established or after
an error occurs.
            public void run() {
                try {
                    addToTranscript("Connecting to " + host
+ " ...");
                    connection = new ChatClient(host);
                    connected = true;
                    Platform.runLater( () -> {
                        messageInput.setEditable(true);
                        messageInput.setDisable(false);
                        sendButton.setDisable(false);
```

```
                        messageInput.requestFocus();
                    });
                }
                catch (IOException e) {
                    Platform.runLater( () -> {
                        addToTranscript("Connection attempt
failed.");
                        addToTranscript("Error: " + e);
                    });
                }
            }
        }.start();

    }



    /**
     * A ChatClient connects to the Hub and is used to send
messages to
     * and receive messages from a Hub.  Four types of
message are
     * received from the Hub.  A ForwardedMessage
represents a message
     * that was entered by some user and sent to all users
of the
     * chat room.  A PrivateMessage represents a message
that was
     * sent by another user only to this user.  A
ClientConnectedMessage
     * is sent when a new user enters the room.  A
ClientDisconnectedMessage
     * is sent when a user leaves the room.
     */
    private class ChatClient extends Client {

        /**
         * Opens a connection 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.
         */
```

```
        protected void messageReceived(Object message) {
            if (message instanceof ForwardedMessage) {
                ForwardedMessage fm =
(ForwardedMessage)message;
                String senderName =
clientNameMap.get(fm.senderID);
                addToTranscript(senderName + " SAYS:  " +
fm.message);
            }
            else if (message instanceof PrivateMessage) {
                PrivateMessage pm =
(PrivateMessage)message;
                String senderName =
clientNameMap.get(pm.senderID);
                addToTranscript("PRIVATE MESSAGE FROM " +
senderName + ":  " + pm.message);
            }
            else if (message instanceof
ClientConnectedMessage) {
                ClientConnectedMessage cm =
(ClientConnectedMessage)message;
                addToTranscript('"' +
cm.nameMap.get(cm.newClientID) + "\" HAS JOINED THE CHAT
ROOM.");
                newNameMap(cm.nameMap);
            }
            else if (message instanceof
ClientDisconnectedMessage) {
                ClientDisconnectedMessage dm =
(ClientDisconnectedMessage)message;
                addToTranscript('"' +
clientNameMap.get(dm.departingClientID) + "\" HAS LEFT THE
CHAT ROOM.");
                newNameMap(dm.nameMap);
            }
        }

        /**
         * This method is part of the connection set up.
It sends the user's selected
         * name to the hub by writing that name to the
output stream.  The hub will
         * respond by sending the name back to this client,
possibly modified if someone
         * is the chat room is already using the selected
name.
```

```
         */
        protected void extraHandshake(ObjectInputStream in,
ObjectOutputStream out) throws IOException {
            try {
                out.writeObject(myName);  // Send user's
name request to the server.
                myName = (String)in.readObject();  // Get
the actual name from the server.
            }
            catch (Exception e) {
                throw new IOException("Error while setting
up connection: " + e);
            }
        }

        /**
         * 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);
            Platform.runLater( () -> {
                sendButton.setDisable(true);
                messageInput.setDisable(true);
                messageInput.setEditable(false);
                messageInput.setText("");
                sendPrivateButton.setDisable(true);
                privateMessageInput.setDisable(true);
                privateMessageInput.setEditable(false);
            });
            connected = false;
            connection = null;
        }

        // Note:  the methods playerConnected() and
playerDisconnected(), which where present here
        // in ChatRoomWindow, were removed, since their
functionality (to announce arrivals
        // and departures) has been taken over by
ClientConnectedMessage and ClientDisconnectedMessage.

    } // end nested class ChatClient
```

```
    /**
     * Adds a string to the transcript area, followed by a
blank line.
     */
    private void addToTranscript(String message) {
        Platform.runLater( () ->
transcript.appendText(message + "\n\n") );
    }


    /**
     * Called when the user clicks the Quit button or
closes
     * the window by clicking its close box. Called from
the
     * application thread.
     */
    private void doQuit() {
        if (connected)
            connection.disconnect();  // Sends a
DisconnectMessage to the server.
        try {
            Thread.sleep(500); // Time for
DisconnectMessage to actually be sent.
        }
        catch (InterruptedException e) {
        }
        System.exit(0);
    }

    /**
     * This method is called when a ClientConnectedMessage
or ClientDisconnectedMessage
     * is received from the hub.  Its job is to save the
nameMap that is part of the
     * message and use it to rebuild the contents of the
ComboBox, clientList, where
     * the user selects the recipient of a private message.
It also enables or
     * disables the private message input box and send
button, depending on whether
     * there are any possible message recipients.
     * @param nameMap the new nameMap, which will replace
the value of clientNameMap.
     */
    private void newNameMap(final TreeMap<Integer,String>
```

```
nameMap) {
        Platform.runLater( () ->  {
            clientNameMap = nameMap;
            String currentlySelected =
clientList.getSelectionModel().getSelectedItem();
            clientList.getItems().clear();
            boolean someoneIsThere = false;
            boolean currentSelectionIsThere = false;
            for (String str: nameMap.values()) {
                if (!str.equals(myName)) {
                    clientList.getItems().add(str);
                    someoneIsThere = true;
                }
                if (str.equals(currentlySelected))
                    currentSelectionIsThere = true;
            }

privateMessageInput.setEditable(someoneIsThere);

privateMessageInput.setDisable(!someoneIsThere);
            sendPrivateButton.setDisable(!someoneIsThere);
            if (!someoneIsThere)
                clientList.getItems().add("(no one
available)");
            if (currentSelectionIsThere)

clientList.getSelectionModel().select(currentlySelected);
            else
                clientList.getSelectionModel().select(0);
        });
    }


    /**
     * Send the string entered by the user as a message
     * to the Hub, using the ChatClient that handles
communication
     * for this ChatRoomWindow.  Note that the string is
not added
     * to the transcript here.  It will get added after the
Hub
     * receives the message and broadcasts it to all
clients,
     * including this one.  Called from the application
thread.
     */
```

```
    private void doSend() {
        String message = messageInput.getText();
        if (message.trim().length() == 0)
            return;
        connection.send(message);
        messageInput.selectAll();
        messageInput.requestFocus();
    }


    private void doSendPrivateMessage() {
        // Send a private message to a specified recipient.
        // If the private message inputbox is empty,
nothing is done.
        String message = privateMessageInput.getText();
        if (message.trim().length() == 0)
            return;
        String recipient =
clientList.getSelectionModel().getSelectedItem(); // name
of recipient.
        int recipientID = -1;  // The ID number of the
recipient
        for (int id : clientNameMap.keySet()) {
            // Search the clientNameMap to find the ID
number
            // corresponding to the specified recipient
name.
            if (recipient.equals(clientNameMap.get(id))) {
                recipientID = id;
                break;
            }
        }
        if (recipientID == -1) {
            Alert alert = new Alert(Alert.AlertType.ERROR,
                    "Funny... The selected
recipient\ndoesn't seem to exit???");
            alert.showAndWait();
            return;
        }
        connection.send(new
PrivateMessage(recipientID,message));
        addToTranscript("Sent to " + recipient + ":  " +
message);
    }
```

```
} // end class NewChatRoomWindow
```

Solution for Programming Exercise 13.1

Exercise 13.1:

The folder nature-images contains several pictures of animals. (In the web site download, you can find that folder in the `chapter13` directory inside the `source` directory.) Write a "scratch off" program that could be used by a small child that works as follows: The program window starts by showing a large uniformly colored rectangle, with one of the animal pictures hidden behind it. As the user drags the mouse over the image, part of the colored overlay is scratched off, revealing the picture underneath. Here is what it should look like after part of the overlay has been removed:

You can implement this by using one canvas, containing the colored overlay, stacked on top of another canvas that contains the animal picture. (Stacked canvases were used in the sample program ToolPaint.java from Subsection 13.2.4.) To implement scratching off part of the overlay, just clear a small rect in the overlay canvas around the mouse location. The program should have some way to move on to the next picture. Another idea is to have

several different sizes of scratchers, so that an impatient child can use a giant one that will remove large swatches of color.

Discussion

The pictures have to be read into the program as objects of type *Image*. I store the pictures into an `ArrayList<Image>`, `pictures`. I shuffle them into a random order using the standard function `Collections.shuffle()`, so that the pictures will be used in a different order each time the program is run. An instance variable, `nextPicIndex`, is used to keep track of which picture is going to be shown next.

The basic stacked canvas idea is easy to implement. The two canvases are placed into a *StackPane*. An animal picture is drawn in the bottom canvas. The animal pictures are not all the same size, so I picked an arbitrary canvas size of 800-by-600 and stretched the pictures to fit the canvas, using a `drawImage()` method that includes the width and height of the rectangle into which the image is drawn. At the same time, I fill the overlay canvas with a random color:

```
overlayGraphics.setFill( Color.hsb(360*Math.random(), 0.5,
0.5) );
overlayGraphics.fillRect( 0,0, width,height );
picGraphics.drawImage( pictures.get(nextPicIndex), 0,0,
width,height );
```

I added some controls at the bottom of the program window: a button that loads the next picture, a *ComboBox* for selecting the size of the "scratcher," and (for the really impatient), a button that will clear the entire overlay canvas.

The Solution

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
```

```
import javafx.scene.control.ComboBox;
import javafx.scene.layout.HBox;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.StackPane;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.paint.Color;
import javafx.scene.image.Image;
import javafx.scene.input.MouseEvent;
import javafx.geometry.Pos;
import javafx.geometry.Insets;

import java.util.ArrayList;
import java.util.Collections;


/**
 * A program that lets the user "scratch off" a mask that
is covering an
 * image, by dragging the mouse.  This program requires
that some specific
 * image files be available as resource files in a folder
named "nature-images".
 */
public class ScratchOff extends Application {

    public static void main(String[] args) {
        launch();
    }
    //-------------------------------------------------------
--------------

    private final static String[] picFileNames = {
                // These are the names of the resource
image files that contain
                // the pictures used by this program.  The
must be in folder
                // named nature-images.
            "bluejay.jpg", "chipmunk.jpg", "collie.jpg",
"elephants.jpg",
            "faun.jpg", "lion.jpg", "polar-bear.jpg",
"stork.jpg"
    };

    private Canvas picCanvas;                // canvas
where picture is drawn
    private GraphicsContext picGraphics;     // graphics
context for picCanvas
```

```
    private Canvas overlayCanvas;            // canvas that
hides the picture
    private GraphicsContext overlayGraphics; // graphics
context for overlayCanvas

    private ArrayList<Image> pictures;  // An arraylist
containing the images
    private int nextPicIndex;  // The index of the array
list of the next picture
                               // that will be drawn.  To
avoid repeating a picture
                               // until all of the pictures
have been used,
                               // the array list is
shuffled and the pictures are
                               // shown in the reverse
order that they occur in the
                               // list.  If all the
pictures are used, the list is
                               // reshuffled and the
pictures are reused.

    private final int[] sizes = { 10, 20, 35, 60 };  //
sizes for the square that
                                                     // is
revealed when the user
                                                     //
drags the mouse
    private double scratcherSize; // current size of the
rectangle; one of the
                                  // items from the
previous array, depeding on
                                  // the current setting of
a ComboBox

    private final static int width = 800;    // canvas size
    private final static int height = 600;

    /**
     * Set up the GUI.  The overlay canvas, which hides the
picture is
     * stacked on top of the canvas that displays the
image.  There are
     * some buttons and a combo box at the bottom of the
window.
     */
    public void start(Stage stage) {
```

```
        picCanvas = new Canvas(width,height);
        picGraphics = picCanvas.getGraphicsContext2D();
        overlayCanvas = new Canvas(width,height);
        overlayGraphics =
overlayCanvas.getGraphicsContext2D();

        pictures = new ArrayList<Image>();
        for (int i = 0; i < picFileNames.length; i++) {
            pictures.add( new Image("nature-images/" +
picFileNames[i]) );
        }
        Collections.shuffle(pictures);  // put pictures in
a random order
        nextPicIndex = picFileNames.length - 1;  // index
for the first picture
        newPic();  // Set up the first picture.

        overlayCanvas.setOnMousePressed( this::scratch );
        overlayCanvas.setOnMouseDragged( this::scratch );

        StackPane canvasHolder = new
StackPane(picCanvas,overlayCanvas);
        canvasHolder.setStyle("-fx-border-color: #444; -fx-
border-width:5px");

        Button newPicBtn = new Button("New ScratchOff!");
        newPicBtn.setOnAction( e -> newPic() );
        Button showAllBtn = new Button("Show the Whole
Picture!");
        showAllBtn.setOnAction( e ->
overlayGraphics.clearRect(0,0,width,height) );

        ComboBox<String> sizeSelect = new ComboBox<>();
        sizeSelect.getItems().addAll("Small", "Normal",
"Big", "Giant");
        sizeSelect.getSelectionModel().select(1);
        scratcherSize = sizes[1];
        sizeSelect.setOnAction(
                e -> scratcherSize = sizes[
sizeSelect.getSelectionModel().getSelectedIndex() ] );

        HBox bottom = new HBox(10,newPicBtn,showAllBtn,
new Label("  Scratcher Size:"), sizeSelect);
        bottom.setAlignment(Pos.CENTER);
        bottom.setPadding( new Insets(10) );

        BorderPane root = new BorderPane(canvasHolder);
        root.setBottom(bottom);
```

```
        stage.setScene( new Scene(root) );
        stage.setResizable(false);
        stage.setTitle("Scratc Off to Reveal a Picture!");
        stage.show();

    }

    /**
     * Show the next picture in the picCanvas, and fill the
overlay canvas
     * with a random dull color to hide the picture.  Set
up nextPicIndex
     * for the next picture.
     */
    private void newPic() {
        overlayGraphics.setFill(
Color.hsb(360*Math.random(), 0.5, 0.5) );
        overlayGraphics.fillRect(0,0,width,height);

picGraphics.drawImage(pictures.get(nextPicIndex),0,0,width,
height);
        if (nextPicIndex == 0) {
            Image item0 = pictures.get(0);
            do { // Avoid having the item that is showing
now be put
                 // at the end of the list where it will be
shown next.
                Collections.shuffle(pictures);
            } while (pictures.get(pictures.size()-1) ==
item0);
            nextPicIndex = pictures.size() - 1;
        }
        else {
            nextPicIndex--;
        }
    }

    /**
     * When the user pressed or drags the mouse, clear a
rect in the
     * overlay canvas, with a size given by scratcherSize.
     */
    private void scratch(MouseEvent evt) {
        overlayGraphics.clearRect( evt.getX() -
scratcherSize/2,
                evt.getY() - scratcherSize/2,
scratcherSize, scratcherSize );
```

```
    }

} // end ScratchOff
```

Solution for Programming Exercise 13.2

Exercise 13.2:

The *StopWatchLabel* component from Subsection 13.3.1 displays the text "Timing..." when the stopwatch is running. It would be nice if it displayed the elapsed time since the stopwatch was started. For that, you need to create an *AnimationTimer*. (See Subsection 6.3.5.) Add an *AnimationTimer* to the original source code, StopWatchLabel.java, to drive the display of the elapsed time in seconds.

Discussion

This one is really too easy! As suggested, I added an *AnimationTimer*, named `timer`, as an instance variable. (I also changed some of the occurrences of the word "timer" in the original program to "stopwatch" to try to avoid some confusion: the stopwatch is a kind of timer, and we are using another kind of timer to implement it.) When the stopwatch starts running, it is necessary to call `timer.start()` to start the animation timer running, and when the stopwatch stops running, `timer.stop() must be called.` The only other thing that is needed is to create the *AnimationTimer* object. Since *AnimationTimer* is an abstract class, we need a subclass, and we need to provide a definition for the abstract method `handle()`. That method, which will be called many times per second, can update the label to show the elapsed time. Using an anonymous class,

```
timer = new AnimationTimer() {
    public void handle(long now) {
        long elapsedTime = System.currentTimeMillis() -
startTime;
        String text = String.format(
                 "%3.1f seconds elapsed",
elapsedTime/1000.0);
        setText(text);
```

```
        }
    };
```

In the program, I don't create the animation timer until the first time the stopwatch is started. Note that when the stopwatch is stopped, the program displays the time with three digits after the decimal point. While it's running, it doesn't make sense to be that accurate since the display won't be updated every millisecond. So, I only use one digit after the decimal point when the stopwatch is running.

The Solution

The timer component, with changes from the original shown in red italic:

```
import javafx.scene.control.Label;
import javafx.animation.AnimationTimer;

/**
 * A custom component that acts as a simple stopwatch.
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
 * displays the number of seconds that have passed since
the stop
 * watch was started.
 */
public class StopWatchLabel2 extends Label {

    private long startTime;   // Start time of timer.
                              //   (Time is measured in
milliseconds.)

    private boolean running;  // True when the timer is
running.

    private AnimationTimer timer;  // Used to update the
timer to show
                                   // the number of seconds
```

```
that have passed.

    /**
     * Constructor sets initial text on the label to
     * "Click to start timer." and sets up a mouse event
     * handler so the label can respond to clicks.
     */
    public StopWatchLabel2() {
        super("  Click to start timer.  ");
        setOnMousePressed( e -> setRunning( !running ) );
    }


    /**
     * Tells whether the timer is currently running.
     */
    public boolean isRunning() {
        return running;
    }


    /**
     * Sets the stop watch to be running or stopped, and
changes the text that
     * is shown on the label.  (This method should be
called on the JavaFX
     * application thread.)
     * @param running says whether the stopwatch should be
running; if this
     *    is equal to the current state, nothing is done.
     */
    public void setRunning( boolean running ) {
        if (this.running == running)
            return;
        this.running = running;
        if (running == true) {
                // Record the time and start the stopwatch.
            startTime = System.currentTimeMillis();
            if (timer == null) {
                timer = new AnimationTimer() {
                    public void handle(long now) {
                        long elapsedTime =
System.currentTimeMillis() - startTime;
                        String text = String.format(
                                 "%3.1f seconds elapsed",
elapsedTime/1000.0);
                        setText(text);
```

```
                    }
                };
            }
            setText("   0.0 seconds elapsed");
            timer.start();
        }
        else {
                // Stop the stopwatch.  Compute the elapsed
time since the
                // stopwatch was started and display it.
            running = false;
            timer.stop();
            long endTime = System.currentTimeMillis();
            double seconds = (endTime - startTime) /
1000.0;
            setText( String.format("Time: %1.3f seconds",
seconds) );
        }
    }

} // end StopWatchLabel
```

The code for the program that tests the component:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.layout.StackPane;
import javafx.geometry.Pos;

/**
 * Shows a StopWatchLabel2.  The user can click to start a
timer
 * and click again to stop it.  The elapsed time is
displayed.
 */
public class TestStopWatch2 extends Application {

    public static void main(String[] args) {
        launch(args);
    }
    //-------------------------------------------------------
-------

    public void start(Stage stage) {
```

```
        StopWatchLabel2 stopWatch = new StopWatchLabel2();
        stopWatch.setStyle("-fx-font: bold 30pt serif; -fx-
background-color:#ffffee;"
                + "-fx-border-color:#008; -fx-border-
width:3px; -fx-padding:20px;"
                + "-fx-text-fill: #008");

stopWatch.setMaxSize(Double.POSITIVE_INFINITY,Double.POSITI
VE_INFINITY);
        stopWatch.setAlignment(Pos.CENTER);

        stage.setScene( new Scene(new StackPane(stopWatch))
);
        stage.setTitle("StopWatchLabel Demo");
        stage.show();

    }

}
```

Solution for Programming Exercise 13.3

Exercise 13.3:

Improve the program ToolPaint.java, from Section 13.2. You can any improvements you like, but here are some suggestions:

- Have separate menus for "Fill Color" and "Stroke Color".
- Make it possible for the user to draw shapes that are both filled and stroked. For example, add two new tools, "Stroked Filled Rect" and "StrokedFilledOval".
- Add a "Line Width" menu.
- Add keyboard accelerators for some commands (see Subsection 13.5.4).
- Make it possible to use a translucent fill color. A simple approach to this is to use a *CheckMenuItem* to select either fully opaque or 50% opaque fill. I don't advise trying to implement translucent stroke colors, since that's more difficult.
- Add an "Undo" command that will restore the image to what it was before the last time it was modified. This can be implemented by making a copy of the image before you modify it. It's possible to have a multi-level undo, but that's harder and uses more memory.

Remember that the `ToolPaint` program requires SimpleDialogs.java.

Discussion

My solutions follows the suggestions for the features to be added. The individual changes are not too difficult, but there are a lot of them!

To have separate stroke and fill colors, I added a new instance variable, `currentStrokeColor` to represent the stroke color. (I use the original variable, `currentColor` as the fill color; I probably should have renamed it `currentFillColor`.) I renamed the "Color" menu to "Fill Color" and I made

a new "Stroke Color" menu containing the same set of colors, including a "Custom Color" command. This was easy to do: I just copied and edited the code that made the "Color" menu, omitting the code for the commands having to do with the background color.

The "Stroke Color" menu also contains a *CheckMenuItem* "Translucent Fill," which is unchecked by default. When the user selects this command from the menu, the value of a global boolean variable, `translucentFill`, is set to match the value of the checkbox. In the `mousePressed()` routine, the graphics contexts for the image canvas and the overlay canvas have to be configured to use the current stroke and fill colors. For the fill color, the `currentColor` variable always stores a fully opaque color. If translucent fill is turned on, the actual fill color should have the same RGB components as `currentColor`, but its alpha component has to be set to 0.5 for 50% transparency. (In fact, in the end, I decided that 40% opaqueness looked better than 50%, so I used an alpha of 0.4.) Here is the code from `mousePressed` that sets up the graphics contexts, including both the stroke and fill colors and a `currentLineWidth` that is also new in the program:

```
Color fill; // currentColor, possibly with alpha set to
0.4 for a translucent fill.
if (translucentFill)
    fill = Color.color(currentColor.getRed(),
currentColor.getGreen(),

currentColor.getBlue(), 0.4);
else
    fill = currentColor;
canvasGraphics.setFill(fill);
canvasGraphics.setStroke(currentStrokeColor);
canvasGraphics.setLineWidth(currentLineWidth);
overlayGraphics.setFill(fill);
overlayGraphics.setStroke(currentStrokeColor);
overlayGraphics.setLineWidth(currentLineWidth);
```

To control the value of the new `currentLineWidth` variable, I added a "Stroke Width" menu. The commands are created and implemented similarly to the commands in the "Color" menu:

```
int[] lineWidths = { 1, 2, 3, 4, 5, 7, 10, 12, 15, 20 };
ToggleGroup lineWidthGroup = new ToggleGroup();
for (int i = 0; i < lineWidths.length; i++) {
```

```
    RadioMenuItem item = new RadioMenuItem("Width = " +
lineWidths[i]);
    item.setUserData(lineWidths[i]);
    item.setToggleGroup(lineWidthGroup);
    lineWidthMenu.getItems().add(item);
    if (i == 2)
        item.setSelected(true);
}
lineWidthGroup.selectedTogglePropertyProperty().addListener( e ->
{
    Toggle t = lineWidthGroup.getSelectedToggle();
    if (t != null) {
        int width = (Integer)t.getUserData();
        currentLineWidth = width;
    }
});
```

I didn't add a "Custom Stroke Width" command. Perhaps I should have.

I added the two new Tools, "Stroked Filled Rect" and "Stroked Filled Oval." They had to be added to the *Tools* enum as well as to the "Tools" menu. The basic method for drawing shapes, `putCurrentShape(g)`, also needed cases for drawing the new tools:

```
case STROKED_FILLED_OVAL:
    putOval(g,true,startX, startY, currentX, currentY);
    putOval(g,false,startX, startY, currentX, currentY);
    break;
case STROKED_FILLED_RECT:
    putRect(g,true,startX, startY, currentX, currentY);
    putRect(g,false,startX, startY, currentX, currentY);
    break;
```

The second parameter to `putOval()` and `putRect()` tells whether to stroke the shape or fill it. By calling the method twice, with the parameter equal to `true` in one call and `false` in the other, the shape is both filled and stroked. Adding these two cases to `putCurrentShape()` was really the only thing necessary to draw the two new shapes.

For the new "Undo" command, I used a simple one-level undo, so that the program just has to save the most recent previous image. I added an "Edit" menu to the program to hold the new command. (To avoid having just one item in that menu, I moved the three background color commands from the

"Color" menu to the new "Edit" menu.) To implement the "Undo" command, I needed to save the current image before modifying it in any way. To get a copy of the current picture in the canvas, I made a "snapshot" of the canvas. I wrote a short method to do that, and to save the snapshot in a global variable, `imageForRedo`, of type *Image*:

```
/**
 * Save a copy of the current image for the Undo
operation. This is
 * called before making any change to the image.
 */
private void saveImageForUndo() {
    imageForUndo = canvas.snapshot(null,null);
    undoItem.setDisable(false); // Undo is now possible.
}
```

This method must be called at every point in the code where the picture in the canvas is about to be modified. Of course, I didn't want to save an image unless the picture actually was modified. For example, in the original program, when drawing a rect or an oval, it is possible that when the drag gesture ends, the rectangle or oval is empty. The original program draws the empty figure into the picture, even though doing so doesn't actually draw anything. In the new program, I check whether the rect or oval is empty before drawing it; if it is empty, I don't want to call `saveImageForRedo()`. Similarly, in `doOpenImage()`, I don't call `saveImageForRedo()` until I am sure that the new image has been successfully loaded into the program.

The method that is called in response to the "Undo" command draws the saved image into the display canvas. Of course, before doing that, it needs to save the current image in case the user wants to "undo the undo," and the method needs to be careful to draw the old saved image into the picture, not the new one:

```
/**
 * Implements the "Undo" command.
 */
private void doUndo() {
    if (imageForUndo == null)
        return;
    Image previousUndoImage = imageForUndo; //Keep the
current undo image.
    saveImageForUndo();  // Save image that is about to be
```

```
replaced, for "undoing the undo".
    canvasGraphics.drawImage(previousUndoImage, 0, 0);  //
Replace image with previous image.
}
```

Finally, adding accelerators to menu commands is straightforward. I added an accelerator for every tool, as well as accelerators for the commands in the "File" menu. The only hard part was deciding what letters to use for all the different accelerators.

The Solution

Significant changes from the original [ToolPaint] program are shown in red italic

```
import java.awt.image.BufferedImage;
import java.io.File;
import javax.imageio.ImageIO;

import javafx.application.Application;
import javafx.embed.swing.SwingFXUtils;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.StackPane;
import javafx.scene.Scene;
import javafx.stage.FileChooser;
import javafx.stage.Stage;
import javafx.scene.control.MenuBar;
import javafx.scene.control.Alert;
import javafx.scene.control.Menu;
import javafx.scene.control.MenuItem;
import javafx.scene.control.SeparatorMenuItem;
import javafx.scene.control.CheckMenuItem;
import javafx.scene.control.Toggle;
import javafx.scene.control.RadioMenuItem;
import javafx.scene.control.ToggleGroup;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.paint.Color;
import javafx.scene.input.MouseEvent;
import javafx.scene.input.KeyCombination;
import javafx.scene.shape.StrokeLineCap;
```

```
import javafx.scene.image.WritableImage;
import javafx.scene.image.Image;
import javafx.scene.image.PixelReader;
import javafx.scene.image.PixelWriter;
import javafx.scene.SnapshotParameters;
import javafx.geometry.Rectangle2D;

/**
 * A simple paint program that lets the user paint with
several
 * different tools, including a "smudge" tool that uses
fairly
 * sophisticated pixel manipulation.  The program also
demonstrates
 * using a transparent "overlay" canvas to implement some
of the
 * tools.  The window for this program is not resizable.
 */
public class ToolPaintPlus extends Application {

    public static void main(String[] args) {
        launch(args);
    }

    private Canvas canvas; // The canvas on which the image
is drawn.
    private GraphicsContext canvasGraphics;  // The
graphics context for the canvas.

    private Canvas overlay; // A transparent canvas that
lies on top of
                            // the image canvas, used for
temporarily
                            // drawing some shapes during a
mouse drag.
    private GraphicsContext overlayGraphics;  // Graphics
context for overlay.


    /**
     * The possible drawing tools in this program.  (The
CURVE tool allows the
     * user to sketch a free-hand curve, while the LINE
tool draws a line
     * between two points.  The SMUDGE tool lets the user
"spread paint around"
     * with the mouse.  The ERASE tool erases with a 10-by-
10 pixel rectangle.)
```

```
     */
    private enum Tool { LINE, RECT, OVAL, FILLED_RECT,
FILLED_OVAL,
                            STROKED_FILLED_RECT,
STROKED_FILLED_OVAL, CURVE, SMUDGE, ERASE }


    private Tool currentTool = Tool.CURVE;  // The current
drawing tool.

    private Color currentColor = Color.BLACK; // The
current fill color.  This color
                                              // is fully
opaque.
    private boolean translucentFill;  // When this is true,
the fill color that is
                                      //   used takes its
RGB components from the
                                      //   currntColor, but
its alpha component is 0.4.

    private Color currentStrokeColor = Color.BLACK;   //
The current stroke color.

    private Color backgroundColor = Color.WHITE;  // The
current background color.

    private double currentLineWidth = 2;  // The line width
used for all strokes.

    private MenuItem undoItem;   // The "Undo" menu item.
This is needed so that
                                 // it can be enabled when
an undo becomes available.
    private Image imageForUndo;  // Stores a snapshot of
the canvas before the
                                 // most recent change to
the image.  This is used
                                 // to implement the "Undo"
command.  Note that
                                 // using Undo twice in a
row will "undo the undo."

    /* Some variables that are used during dragging. */
    private boolean dragging;     // is a drag in progress?
    private int startX, startY;   // start point of drag
```

```
    private int prevX, prevY;     // previous mouse
location during a drag
    private int currentX, currentY;  // current mouse
position during a drag
    private boolean firstMove;    // used in mouseDragged
to tell whether this
                                  //    is the first time
the mouse has moved.

    /* Some variables used to implement the smudge tool. */
    private double[][] smudgeRed, smudgeGreen, smudgeBlue;
    private WritableImage pixels; // a 9-by-9 block of
pixels from the canvas
    private PixelReader pixelReader;  // for reading colors
from pixels
    private SnapshotParameters snapshotParams;  // used for
snapshotting the canvas
    private PixelWriter pixelWriter;  // for writing pixels
to the canvas

    private Stage window;  // The program's window.

    /**
     * Create the canvas and the overlayCanvas, and set up
mouse handling,
     * configure and show the window.
     */
    public void start(Stage stage) {

        window = stage;

        int width = 800;   // size of canvas; can be
changed here
        int height = 600;

        canvas = new Canvas(width,height);
        canvasGraphics = canvas.getGraphicsContext2D();
        canvasGraphics.setFill(backgroundColor);
        canvasGraphics.fillRect(0,0,width,height);
        overlay = new Canvas(width,height);
        overlayGraphics = overlay.getGraphicsContext2D();
        overlay.setOnMousePressed( e -> mousePressed(e) );
        overlay.setOnMouseDragged( e -> mouseDragged(e) );
        overlay.setOnMouseReleased( e -> mouseReleased(e)
);
        // canvasGraphics.setLineWidth(2);  // line width
is now set in mousePressed
```

```
        // overlayGraphics.setLineWidth(2);
        canvasGraphics.setLineCap(StrokeLineCap.ROUND);  //
will look better for thick strokes
        overlayGraphics.setLineCap(StrokeLineCap.ROUND);

        StackPane canvasHolder = new
StackPane(canvas,overlay);
        BorderPane root = new BorderPane(canvasHolder);
        root.setTop( makeMenuBar() );

        stage.setScene( new Scene(root) );
        stage.setTitle("A Simple Paint Program");
        stage.setResizable(false);
        stage.show();

    } // end start()


    /**
     * A utility method to draw the current shape in a
given graphics context.
     * It draws the correct shape for the current tool in a
rectangle whose
     * corners are given by the starting position of the
mouse and the current
     * position of the mouse.  This method is not used when
the current tool
     * is Tool.CURVE or Tool.ERASE, or Tool.SMUDGE.  For
other tools, it is
     * used to draw the shape to the overlay canvas during
a drag operation;
     * then, when the drag ends, it is used to draw the
shape to the main
     * canvas.  The shape is defined by the tool and by the
two points
     * (startX,startY) and (currentX,currentY).
     */
    private void putCurrentShape(GraphicsContext g) {
        switch (currentTool) {
        case LINE:
            if (startX != currentX || startY != currentY)
                g.strokeLine(startX, startY, currentX,
currentY);
            break;
        case OVAL:
            putOval(g,false,startX, startY, currentX,
currentY);
```

```
            break;
        case RECT:
            putRect(g,false,startX, startY, currentX,
currentY);
            break;
        case FILLED_OVAL:
            putOval(g,true,startX, startY, currentX,
currentY);
            break;
        case FILLED_RECT:
            putRect(g,true,startX, startY, currentX,
currentY);
            break;
        case STROKED_FILLED_OVAL:
            putOval(g,true,startX, startY, currentX,
currentY);
            putOval(g,false,startX, startY, currentX,
currentY);
            break;
        case STROKED_FILLED_RECT:
            putRect(g,true,startX, startY, currentX,
currentY);
            putRect(g,false,startX, startY, currentX,
currentY);
            break;
        default:  // not called in other cases
            break;
        }
    }


    /**
     * Draws a filled or unfilled rectangle with corners at
the points (x1,y1)
     * and (x2,y2).  Nothing is drawn if x1 == x2 or y1 ==
y2.
     * (This method translates from an opposite-corners
definition of the rectangle
     * to the upper-left-corner-width-and-height definition
used for drawing.)
     * @param g the graphics context where the rectangle is
drawn
     * @param filled tells whether to draw a filled or
unfilled rectangle.
     */
    private void putRect(GraphicsContext g, boolean filled,
int x1, int y1, int x2, int y2) {
```

```
        if (x1 == x2 || y1 == y2)
            return;
        int x = Math.min(x1,x2);    // get upper left
corner, (x,y)
        int y = Math.min(y1,y2);
        int w = Math.abs(x1 - x2);  // get width and height
        int h = Math.abs(y1 - y2);
        if (filled)
            g.fillRect(x,y,w,h);
        else
            g.strokeRect(x,y,w,h);
    }


    /**
     * Draws a filled or unfilled oval in the rectangle
with corners at the
     * points (x1,y1) and (x2,y2).  Nothing is drawn if x1
== x2 or y1 == y2.
     * @param g the graphics context where the oval is
drawn
     * @param filled tells whether to draw a filled or
unfilled oval.
     */
    private void putOval(GraphicsContext g, boolean filled,
int x1, int y1, int x2, int y2) {
        if (x1 == x2 || y1 == y2)
            return;
        int x = Math.min(x1,x2);    // get upper left
corner, (x,y)
        int y = Math.min(y1,y2);
        int w = Math.abs(x1 - x2);  // get width and height
        int h = Math.abs(y1 - y2);
        if (filled)
            g.fillOval(x,y,w,h);
        else
            g.strokeOval(x,y,w,h);
    }


    /**
     * Creates a menu bar for use with this program, with
"Color"
     * and "Tool" menus.
     */
    private MenuBar makeMenuBar() {
        MenuBar menubar = new MenuBar();
        Menu fileMenu = new Menu("File");
```

```
        Menu editMenu = new Menu("Edit");
        Menu colorMenu = new Menu("Fill Color");
        Menu strokeColorMenu = new Menu("Stroke Color");
        Menu lineWidthMenu = new Menu("Stroke Width");
        Menu toolMenu = new Menu("Tool");
        menubar.getMenus().add(fileMenu);
        menubar.getMenus().add(editMenu);
        menubar.getMenus().add(toolMenu);
        menubar.getMenus().add(colorMenu);
        menubar.getMenus().add(strokeColorMenu);
        menubar.getMenus().add(lineWidthMenu);

        /* Add items to the File menu. */

        MenuItem openImage = new MenuItem("Load Image...");
        openImage.setOnAction( e -> doOpenImage() );
        openImage.setAccelerator(
KeyCombination.valueOf("shortcut+O"));
        fileMenu.getItems().add(openImage);
        MenuItem saveImage = new MenuItem("Save PNG
Image...");
        saveImage.setOnAction( e -> doSaveImage() );
        saveImage.setAccelerator(
KeyCombination.valueOf("shortcut+S"));
        fileMenu.getItems().add(saveImage);
        fileMenu.getItems().add( new SeparatorMenuItem() );
        MenuItem quit = new MenuItem("Quit");
        quit.setOnAction( e -> System.exit(0) );
        quit.setAccelerator(
KeyCombination.valueOf("shortcut+Q"));
        fileMenu.getItems().add(quit);

        /* Add items to the Edit menu.  (All items except
"Undo" were in the
         * "Color" menu in the previous version.) */

        undoItem = new MenuItem("Undo");
        undoItem.setDisable(true); // will be enabled when
some change is made
        undoItem.setOnAction( e -> doUndo() );
        undoItem.setAccelerator(
KeyCombination.valueOf("shortcut+Z"));
        editMenu.getItems().add(undoItem);
        editMenu.getItems().add( new SeparatorMenuItem() );
        MenuItem clear = new MenuItem("Clear to Background
Color");
        clear.setOnAction( e -> {  // Fill main canvas with
```

```
current background color.
            saveImageForUndo();
            canvasGraphics.setFill(backgroundColor);

canvasGraphics.fillRect(0,0,canvas.getWidth(),canvas.getHei
ght());
        });
        clear.setAccelerator(
KeyCombination.valueOf("shortcut+K") );
        editMenu.getItems().add(clear);
        MenuItem fill = new MenuItem("Fill with Drawing
Color");
        fill.setOnAction( e -> {  // Fill main canvas with
current drawing color, but
                                  // don't change the
background color.  (The erase
                                  // tool will still erase
to the (old) background color.)
            saveImageForUndo();
            canvasGraphics.setFill(currentColor);

canvasGraphics.fillRect(0,0,canvas.getWidth(),canvas.getHei
ght());
        });
        editMenu.getItems().add(fill);
        editMenu.getItems().add(new SeparatorMenuItem());
        MenuItem setBG = new MenuItem("Fill and Set
Background...");
        setBG.setOnAction( e -> {
                 // User can select a new background color
from a dialog box.  If the
                 // dialog box is not canceled, the
selected color becomes the background
                 // color, and the canvas is filled with
that background color.
            Color c =
SimpleDialogs.colorChooser(backgroundColor, "Select a
Background Color");
            if (c != null) {
                saveImageForUndo();
                backgroundColor = c;
                canvasGraphics.setFill(c);

canvasGraphics.fillRect(0,0,canvas.getWidth(),canvas.getHei
ght());
            }
        });
```

```
        setBG.setAccelerator(
KeyCombination.valueOf("shortcut+B") );
        editMenu.getItems().add(setBG);

        /* Color choices are given by RadioMenuItems,
controlled by
         * a ToggleGroup.  Each choice corresponds to a
standard color,
         * except for a "Custom Drawing Color" item that
calls up
         * a color choice dialog box.  The same set of
colors is used
         * in the "Fill Color" menu and in the "Stroke
Color" menu.*/

        Color[] colors = { // Standard colors available in
the menu.
                Color.BLACK, Color.WHITE, Color.RED,
Color.GREEN,
                Color.BLUE, Color.YELLOW, Color.CYAN,
Color.PURPLE, Color.GRAY};
        String[] colorNames = { // Names for the colors,
used to construct menu items.
                "Black", "White", "Red", "Green",
                "Blue", "Yellow", "Cyan", "Purple",
"Gray"};

        /* First, set up the "Fill Color menu. */

        ToggleGroup colorGroup = new ToggleGroup();
        for (int i = 0; i < colors.length; i++) {
            RadioMenuItem item = new RadioMenuItem("Fill
with " + colorNames[i]);
            item.setUserData(colors[i]); // Stash the
actual color in the menu item's UserData
            item.setToggleGroup(colorGroup);
            colorMenu.getItems().add(item);
            if (i == 0)
                item.setSelected(true);  // Initially
selected color is black.
        }
        RadioMenuItem customColor = new
RadioMenuItem("Custom Fill Color...");
        customColor.setToggleGroup(colorGroup);
        customColor.setOnAction( e -> {
               // For the custom color selection, use a
dialog box to get an
```

```
                // arbitrary color from the user.  This has
to be done in an
                // ActionEvent handler, since it needs to
happen even the user selects
                // Custom Drawing Color when it is already
selected.  (In that case,
                // the selected toggle does not change.
            customColor.setSelected(true);
            Color c =
SimpleDialogs.colorChooser(currentColor, "Select a Color to
Use For Filling Shapes");
            if (c != null)  // c is null if user cancels
the dialog
                currentColor = c;
        });
        colorMenu.getItems().add(customColor);
        colorGroup.selectedToggleProperty().addListener( e
-> {
                // Sets the color, when one of the standard
colors is selected.
                // This does not handle the Custom Color
option.  Note that
                // when the user chooses a new radio menu
item, the selected toggle
                // first changes to null as the old menu
item is deselected, then
                // changes to the newly selected menu item.
            Toggle t = colorGroup.getSelectedToggle();  //
the selected RadioMenuItem
            if (t != null && t != customColor) {
                    // The color associated with this menu
item was stashed
                    // in the UserData of the menu item, t.
                currentColor = (Color)t.getUserData();
            }
        });
        colorMenu.getItems().add( new SeparatorMenuItem()
);
        CheckMenuItem translucent = new CheckMenuItem("Use
Translucent Fill");
        translucent.setOnAction( e -> translucentFill =
translucent.isSelected() );
        colorMenu.getItems().add(translucent);

        /* Second, set up the "Stroke Color" menu in a
similar way,
         * but without the extra commands. */
```

```
        ToggleGroup strokeColorGroup = new ToggleGroup();
        for (int i = 0; i < colors.length; i++) {
            RadioMenuItem item = new RadioMenuItem("Stroke
with " + colorNames[i]);
            item.setUserData(colors[i]);
            item.setToggleGroup(strokeColorGroup);
            strokeColorMenu.getItems().add(item);
            if (i == 0)
                item.setSelected(true);  // Initially
selected color is black.
        }
        RadioMenuItem customStrokeColor = new
RadioMenuItem("Custom Stroke Color...");
        customStrokeColor.setToggleGroup(strokeColorGroup);
        customStrokeColor.setOnAction( e -> {
            customStrokeColor.setSelected(true);
            Color c =
SimpleDialogs.colorChooser(currentStrokeColor, "Select a
Color to Use For Strokes");
            if (c != null)
                currentStrokeColor = c;
        });
        strokeColorMenu.getItems().add(customStrokeColor);

strokeColorGroup.selectedToggleProperty().addListener( e ->
{
            Toggle t =
strokeColorGroup.getSelectedToggle();
            if (t != null && t != customStrokeColor) {
                currentStrokeColor =
(Color)t.getUserData();
            }
        });

        /* Set up the "Stroke Width" menu. */

        int[] lineWidths = { 1, 2, 3, 4, 5, 7, 10, 12, 15,
20 };
        ToggleGroup lineWidthGroup = new ToggleGroup();
        for (int i = 0; i < lineWidths.length; i++) {
            RadioMenuItem item = new RadioMenuItem("Width =
" + lineWidths[i]);
            item.setUserData(lineWidths[i]);
            item.setToggleGroup(lineWidthGroup);
```

```
            lineWidthMenu.getItems().add(item);
            if (i == 2)
                item.setSelected(true);
        }

lineWidthGroup.selectedTogglePropertyProperty().addListener( e -> {
            Toggle t = lineWidthGroup.getSelectedToggle();
            if (t != null) {
                int width = (Integer)t.getUserData();
                currentLineWidth = width;
            }
        });

        /* The User selects a drawing tool from the tool
menu.  The menu contains
         * an entry for each available tool.  Tools are
represented by RadioMenuItems,
         * controlled by a ToggleGroup. */

        Tool[] tools = { // The available tools in the
order they appear in the menu.
                Tool.CURVE, Tool.LINE, Tool.RECT,
Tool.OVAL, Tool.FILLED_RECT,
                Tool.FILLED_OVAL, Tool.STROKED_FILLED_RECT,
Tool.STROKED_FILLED_OVAL,
                Tool.SMUDGE, Tool.ERASE };
        String[] toolNames = { // Names for the tools, used
as text in the menu items.
                "Curve", "Line", "Rectangle", "Oval",
"Filled Rectangle",
                "Filled Oval", "Stroked Filled Rect",
"Stroked Filled Oval",
                "Smudge", "Erase" };
        String[] toolAccelerators = { // accelerators for
tool command; will be added to "shortcut+"
                "C", "L", "R", "V", "alt+R",
                "alt+V", "shift+R", "shift+V",
                "M", "E" };

        ToggleGroup toolGroup = new ToggleGroup();
        for (int i = 0; i < tools.length; i++) {
            RadioMenuItem item = new
RadioMenuItem(toolNames[i]);
            item.setUserData(tools[i]);  // Stash the
actual tool in the menu items' UserData
            item.setToggleGroup(toolGroup);
```

```
            item.setAccelerator(
KeyCombination.valueOf("shortcut+" + toolAccelerators[i])
);
            toolMenu.getItems().add(item);
            if (i == 0)
                item.setSelected(true);  // Curve tool is
initially selected
            if (i == 0 || i == 5)  // Separators before and
after the shape tools.
                toolMenu.getItems().add(new
SeparatorMenuItem() );
        }
        toolGroup.selectedTogglePropertyProperty().addListener( e -
> {
            Toggle t = toolGroup.getSelectedToggle();  //
The selected RadioMenuItem
            if (t != null)
                currentTool = (Tool)t.getUserData(); // the
actual tool was stashed in the UserData.
        });

        return menubar;

    } // end makeMenuBar


    /**
     * When the ERASE or SMUDGE tools are used and the
mouse jumps
     * from (x1,y1) to (x2,y2), the tool has to be applied
to a
     * line of pixel positions between the two points in
order to
     * be sure to cover the entire line that the mouse
moves along.
     */
    private void applyToolAlongLine(int x1, int y1, int x2,
int y2) {
        int w = (int)canvas.getWidth();   // (for SMUDGE
only)
        int h = (int)canvas.getHeight();  // (for SMUDGE
only)
        int dist = Math.max(Math.abs(x2-x1),Math.abs(y2-
y1));
            // dist is the number of points along the line
from
            // (x1,y1) to (x2,y2) at which the tool will be
```

```
applied.
        double dx = (double)(x2-x1)/dist;
        double dy = (double)(y2-y1)/dist;
        for (int d = 1; d <= dist; d++) {
                // Apply the tool at one of the points
(x,y) along the
                // line from (x1,y1) to (x2,y2).
            int x = (int)Math.round(x1 + dx*d);
            int y = (int)Math.round(y1 + dy*d);
            if (currentTool == Tool.ERASE) {
                    // Erase a 10-by-10 block of pixels
around (x,y)
                canvasGraphics.fillRect(x-5,y-5,10,10);
            }
            else {
                    // For the SMUDGE tool, blend some of
the color from
                    // the smudgeRed, smudgeGreen, and
smudgeBlue arrays
                    // into the pixels in a 7-by-7 block
around (x,y), and
                    // vice versa.  The effect is to smear
out the color
                    // of pixels that are visited by the
tool.
                snapshotParams.setViewport(new
Rectangle2D(x-4,y-4,9,9));
                canvas.snapshot(snapshotParams, pixels);
                for (int j = 0; j < 9; j++) {
                    int c = x - 4 + j;
                    for (int i = 0; i < 9; i++) {
                        int r = y - 4 + i;
                        if ( r >= 0 && r < h && c >= 0 && c
< w && smudgeRed[i][j] != -1) {
                            Color oldColor =
pixelReader.getColor(j,i);
                            double newRed =
(oldColor.getRed()*0.8 + smudgeRed[i][j]*0.2);
                            double newGreen =
(oldColor.getGreen()*0.8 + smudgeGreen[i][j]*0.2);
                            double newBlue =
(oldColor.getBlue()*0.8 + smudgeBlue[i][j]*0.2);
                            pixelWriter.setColor(c,
r,Color.color(newRed,newGreen,newBlue));
                            smudgeRed[i][j] =
oldColor.getRed()*0.2 + smudgeRed[i][j]*0.8;
                            smudgeGreen[i][j] =
oldColor.getGreen()*0.2 + smudgeGreen[i][j]*0.8;
```

```
                                smudgeBlue[i][j] =
oldColor.getBlue()*0.2 + smudgeBlue[i][j]*0.8;
                            }
                        }
                    }
                }
            }
    } // end applyToolAlongLine


    /**
     * Start a drag operation.
     */
    private void mousePressed(MouseEvent evt) {
        startX = prevX = currentX = (int)evt.getX();
        startY = prevY = currentY = (int)evt.getY();
        dragging = true;
        firstMove = true;

        // make sure we are drawing with the right
properties

        Color fill; // currentColor, possibly with alpha
set to 0.4 for a translucent fill.
        if (translucentFill)
            fill = Color.color(currentColor.getRed(),
currentColor.getGreen(),

currentColor.getBlue(), 0.4);
        else
            fill = currentColor;
        canvasGraphics.setFill(fill);
        canvasGraphics.setStroke(currentStrokeColor);
        canvasGraphics.setLineWidth(currentLineWidth);
        overlayGraphics.setFill(fill);
        overlayGraphics.setStroke(currentStrokeColor);
        overlayGraphics.setLineWidth(currentLineWidth);

        if (currentTool == Tool.ERASE) {
                // Erase a 10-by-10 block around the
starting mouse position.
            canvasGraphics.setFill(backgroundColor);  //
change the color when using erase
            canvasGraphics.fillRect(startX-5,startY-
5,10,10);
        }
        else if (currentTool == Tool.SMUDGE) {
```

```
                // Record the colors in a 7-by-7 block of
pixels around the
                // starting mouse position into the arrays
smudgeRed,
                // smudgeGreen, and smudgeBlue.  These
arrays hold the
                // red, green, and blue components of the
colors.
            if (smudgeRed == null) {
                    // Create all variables needed for
smudge, if not already done.
                pixels = new WritableImage(9,9);
                pixelReader = pixels.getPixelReader();
                snapshotParams = new SnapshotParameters();
                smudgeRed = new double[9][9];
                smudgeGreen = new double[9][9];
                smudgeBlue = new double[9][9];
                pixelWriter =
canvasGraphics.getPixelWriter();
            }
            snapshotParams.setViewport(new
Rectangle2D(startX-4,startY-4,9,9));
            canvas.snapshot(snapshotParams, pixels);
            int h = (int)canvas.getHeight();
            int w = (int)canvas.getWidth();
            for (int j = 0; j < 9; j++) {  // row in the
snapshot
                int r = startY + j - 4;  // the
corresponding row in the canvas
                for (int i = 0; i < 9; i++) {  // column in
the snapshot
                    int c = startX + i - 4;  // the
corresponding column in canvas
                    if (r < 0 || r >= h || c < 0 || c >= w)
{
                            // The point (i,j) is outside
the canvas.
                            // A -1 in the smudgeRed array
indicates that the
                            // corresponding pixel was
outside the canvas.
                        smudgeRed[j][i] = -1;
                    }
                    else {
                        Color color =
pixelReader.getColor(i, j); // get color from snapshot
                        smudgeRed[j][i] = color.getRed();
                        smudgeGreen[j][i] =
```

```
color.getGreen();
                        smudgeBlue[j][i] = color.getBlue();
                    }
                }
            }
        }
    }


    /**
     * Continue a drag operation when the user drags the
mouse.
     * For the CURVE tool, a line is drawn from the
previous mouse
     * position to the current mouse position in the main
canvas.
     * For shape tools like LINE and FILLED_RECT, the shape
is drawn
     * to the overlay canvas after first clearing the
overlay canvas.
     * For the SMUDGE and ERASE tools, the tool is applied
along a
     * line from the previous mouse position to the current
position,
     * on the main canvas.
     */
    private void mouseDragged(MouseEvent evt) {
        if (!dragging)
            return;
        currentX = (int)evt.getX();
        currentY = (int)evt.getY();
        if (currentTool == Tool.CURVE) {
            if (firstMove)
                saveImageForUndo();

canvasGraphics.strokeLine(prevX,prevY,currentX,currentY);
        }
        else if (currentTool == Tool.ERASE || currentTool
== Tool.SMUDGE) {
            if (firstMove)
                saveImageForUndo();

applyToolAlongLine(prevX,prevY,currentX,currentY);
        }
        else  {  // tool is a shape that has to be drawn to
overlay canvas
```

```
overlayGraphics.clearRect(0,0,overlay.getWidth(),overlay.ge
tHeight());
            putCurrentShape(overlayGraphics);
        }
        prevX = currentX;
        prevY = currentY;
        firstMove = false;
    }


    /**
     * Finish a mouse drag operation.  Nothing is done
unless the current tool
     * is a shape tool.  For shape tools, the user's shape
is drawn to the
     * main canvas, making it a permanent part of the
picture, and
     * the overlay canvas, which was used for the shape
during dragging,
     * is cleared.
     */
    private void mouseReleased(MouseEvent evt) {
        dragging = false;
        if (currentTool != Tool.CURVE &&
                currentTool != Tool.ERASE && currentTool !=
Tool.SMUDGE) {
            if (currentTool == Tool.LINE) {
                if (currentX == startX && currentY ==
startY) {
                    // mouse is at starting position; there
is no line to draw.
                    return;
                }
            }
            else if (currentX == startX || currentY ==
startY) {
                // The shape has width=0 or height=0 and so
should not be drawn.
                return;
            }
            saveImageForUndo();
            putCurrentShape(canvasGraphics);

overlayGraphics.clearRect(0,0,overlay.getWidth(),overlay.ge
tHeight());
        }
    }
```

```
    /**
     * Save a copy of the current image for the Undo
operation. This is
     * called before making any change to the image.
     */
    private void saveImageForUndo() {
        imageForUndo = canvas.snapshot(null,null);
        undoItem.setDisable(false);
    }


    /**
     * Implements the "Undo" command.
     */
    private void doUndo() {
        if (imageForUndo == null)
            return;
        Image previousUndoImage = imageForUndo;
        saveImageForUndo();  // Save image that is about to
be replaced, for "undoing the undo".
        canvasGraphics.drawImage(previousUndoImage, 0, 0);
// Replace image with previous image.
    }

    /**
     * Reads an image from a file and draws it to the
canvas,
     * scaling it so it fills the canvas.
     */
    private void doOpenImage() {
        FileChooser fileDialog = new FileChooser();
        fileDialog.setInitialFileName("");
        fileDialog.setInitialDirectory(
                                new File(
System.getProperty("user.home") ) );
        fileDialog.setTitle("Select Image File to Load");
        File selectedFile =
fileDialog.showOpenDialog(window);
        if ( selectedFile == null )
            return;  // User did not select a file.
        Image image = new Image("file:" + selectedFile);
        if (image.isError()) {
            Alert errorAlert = new
Alert(Alert.AlertType.ERROR,
```

```
                    "Sorry, an error occurred while\ntrying
to load the file:\n"
                            +
image.getException().getMessage());
            errorAlert.showAndWait();
            return;
        }
        saveImageForUndo();

canvasGraphics.drawImage(image,0,0,canvas.getWidth(),canvas
.getHeight());
    }


    /**
     * Saves the user's sketch as an image file in PNG
format.
     */
    private void doSaveImage() {
        FileChooser fileDialog = new FileChooser();
        fileDialog.setInitialFileName("imagefile.png");
        fileDialog.setInitialDirectory(
                            new File(
System.getProperty("user.home") ) );
        fileDialog.setTitle("Select File to Save. Name MUST
end with .png!");
        File selectedFile =
fileDialog.showSaveDialog(window);
        if ( selectedFile == null )
            return;  // User did not select a file.
        try {
            Image canvasImage = canvas.snapshot(null,null);
            BufferedImage image =
SwingFXUtils.fromFXImage(canvasImage,null);
            String filename =
selectedFile.getName().toLowerCase();
            if ( ! filename.endsWith(".png")) {
                throw new Exception("The file name must end
with \".png\".");
            }
            boolean hasFormat =
ImageIO.write(image,"PNG",selectedFile);
            if ( ! hasFormat ) { // (this should never
happen)
                throw new Exception( "PNG format not
available.");
            }
```

```
        }
        catch (Exception e) {
            Alert errorAlert = new
Alert(Alert.AlertType.ERROR,
                    "Sorry, an error occurred while\ntrying
to save the image:\n"
                          + e.getMessage());
            errorAlert.showAndWait();
        }
    }



} // end class ToolPaintPlus
```

Solution for Programming Exercise 13.4

Exercise 13.4:

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 XML version from the exercise. Use a *TableView* 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.

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 need to save the data when the user closes the window, which ends the program. To do that, you can add a listener to the program's *Stage* to handle the WindowHidden event. For an example of using that event, the Mandelbrot Viewer program from Section 13.5 uses it to save preferences when the program ends. For an example of creating an editable table, see ScatterPlotTableDemo.java.

(I suggest keeping things simple. You not being asked to write a real phone book application! The point is mostly to make an editable table. My program has text input boxes for name and number, and an "Add" button for adding a new entry containing the input in those boxes. My program always saves the data, whether or not the user has changed it. The interface will be poor: The user has to double-click a cell to edit it and press return to finish the edit and save the new value. It is possible to make a table with a better editing interface, but to do that, you need to write a new *CellFactory* class for the table.)

Discussion

To make a table with two columns holding names and phone numbers, we need a class to represent a row in the table. For an editable table, the individual data values in the class should be stored in observable properties. I wrote a simple nested class, *PhoneEntry*, for that purpose, similar to the *Point* class that was used to represent rows in the `ScatterPlotTableDemo` program:

```
/**
 * A class that represents one phone book entry, with
 * observable properties nameProperty() and
numberProperty()
 * to hold the data for the entry. (Note that the table
must
 * be public and the nameProperty() and numberProperty()
method
 * must be public for this class to work with an editable
table.)
 */
public static class PhoneEntry {
    StringProperty name;
    StringProperty number;
    PhoneEntry(String name, String number) {
        this.name = new SimpleStringProperty(name);
        this.number = new SimpleStringProperty(number);
    }
    public StringProperty nameProperty() {
        return name;
    }
    public StringProperty numberProperty() {
        return number;
    }
}
```

The table can then be created as a variable of type `TableView<PhoneEntry>`. The data for the phone book will be stored in the list of items in the table, which is of type `ObservableList<PhoneEntry>`. The program uses a variable, `phoneEntries`, to refer to that list:

```
ObservableList<PhoneEntry> phoneEntries =
phoneBook.getItems();
```

The two table columns have to be created and configured. Again, this is similar to what was done in `ScatterPlotTableDemo`, except that in this program, the user is allowed to sort and resize the columns And since the values in the cells are *Strings* rather than *Doubles*, no *StringConverter* is needed for the cell factory. The "Name" column is created as follows:

```
TableColumn<PhoneEntry, String> nameColumn = new
TableColumn<>("Name");
nameColumn.setCellValueFactory( new
PropertyValueFactory<PhoneEntry, String>("name") );
nameColumn.setCellFactory(
TextFieldTableCell.forTableColumn() );
nameColumn.setPrefWidth(200);  // (Default size is too
small)
phoneBook.getColumns().add(nameColumn);
nameColumn.setEditable(true);
```

The program has "Add" and "Delete" buttons for adding and deleting phone entries. (This part is actually modeled on EditListDemo.java rather than `ScatterPlotListDemo`.) At first, I tried just adding an empty row when the user clicks "Add" and leaving it up to the user to fill in the name and number in that row, but I found that it was hard even to see the empty row in the table. So instead, I added text input boxes for the new name and number, and the "Add" button adds a new entry containing the data from those input boxes. This also allowed me to check that the two input boxes were both non-empty before adding the entry to the table, so that I could avoid having rows in the table where the name or number was empty.

For the file that stores the phone book data, I used the simple data format from the original program. That program already had code for loading and saving the data, at the beginning and end of its `main()` routine. I copied that code into two instance methods named `loadPhoneBook()` and `savePhoneBook()`, and I modified the code to use data stored in an *ObservableList* instead of in a *Map*. The `savePhoneBook()` method is called by an event handler when the window is closed, as suggested in the exercise:

```
stage.setOnHidden( e -> savePhoneBook(dataFile,
phoneEntries) );
```

The Solution

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.control.Alert;
import javafx.scene.control.Button;
import javafx.scene.control.TextField;
import javafx.scene.control.TableView;
import javafx.scene.control.TableColumn;
import javafx.scene.control.cell.PropertyValueFactory;
import javafx.scene.control.cell.TextFieldTableCell;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.geometry.Insets;
import javafx.beans.property.StringProperty;
import javafx.beans.property.SimpleStringProperty;
import javafx.collections.ObservableList;

import java.io.File;
import java.io.PrintWriter;
import java.io.IOException;
import java.util.Scanner;
import java.util.List;


/**
 * Lets the user edit a list names and phone numbers in a
TableView.
 * The list is read from the file named ".phone_book_demo"
in the
 * user's home directory, and the possibly changed data is
written
 * back to that file when the program ends.  The file uses
a
 * simple format: each line contains a name and a phone
number,
 * separated by "%".  (The format can't correctly handle
names that
 * contain a '%' character, which should not be too
likely.)
 */
public class PhoneDirectoryGUI extends Application {
```

```
    public static void main(String[] args) {
        launch();
    }
    // ----------------------------------------------------
---------------------

    /**
     * The name of the file in which the phone book data is
kept.  The
     * file is stored in the user's home directory.  The
"." at the
     * beginning of the file name means that the file will
be a
     * "hidden" file on Unix-based computers, including
Linux and
     * MacOS.
     */
    private static String DATA_FILE_NAME =
".phone_book_demo";


    /**
     * A class that represents one represents one phone
book entry,
     * with observable properties nameProperty() and
numberProperty()
     * to hold the data for the entry. (Note that the table
must
     * be public and the nameProperty() and
numberProperty() method
     * must be public for this class to work with an
editable table.)
     */
    public static class PhoneEntry {
        StringProperty name;
        StringProperty number;
        PhoneEntry(String name, String number) {
            this.name = new SimpleStringProperty(name);
            this.number = new SimpleStringProperty(number);
        }
        public StringProperty nameProperty() {
            return name;
        }
        public StringProperty numberProperty() {
            return number;
        }
    }
```

```
    /**
     * start() method creates the table, reads the existing
data file if
     * there is one, and sets up the GUI.  It installs a
WindowHidden
     * event handler on the stage that will save the data
when the window
     * is closed.
     */
    public void start(Stage stage) {

        /* Create a dataFile variable of type File to
represent the
         * data file that is stored in the user's home
directory. */

        File userHomeDirectory = new File(
System.getProperty("user.home") );
        File dataFile = new File( userHomeDirectory,
DATA_FILE_NAME );

        /* Create a TableView to hold the data for the
phonebook, and
         * load the existing phonebook data from the file
into the table.
         * The loadPhoneBook method will not return if an
error occurs. */

        TableView<PhoneEntry> phoneBook = new TableView<>
();
        phoneBook.setEditable(true);
        phoneBook.setPrefSize(420,350);
        ObservableList<PhoneEntry> phoneEntries =
phoneBook.getItems();

        loadPhoneBook(dataFile, phoneEntries);
        if (phoneEntries.size() == 0)
            phoneEntries.add( new
PhoneEntry("name","number"));

        /* Define the table columns for the "Name" and
"Phone Number" columns. */

        TableColumn<PhoneEntry, String> nameColumn = new
TableColumn<>("Name");
        nameColumn.setCellValueFactory( new
PropertyValueFactory<PhoneEntry, String>("name") );
```

```
        nameColumn.setCellFactory(
TextFieldTableCell.forTableColumn() );
        nameColumn.setPrefWidth(200);  // (Default size is
too small)
        phoneBook.getColumns().add(nameColumn);
        nameColumn.setEditable(true);

        TableColumn<PhoneEntry, String> numberColumn = new
TableColumn<>("Phone Number");
        numberColumn.setCellValueFactory( new
PropertyValueFactory<PhoneEntry, String>("number") );
        numberColumn.setCellFactory(
TextFieldTableCell.forTableColumn() );
        numberColumn.setPrefWidth(200);  // (Default size
is too small)
        phoneBook.getColumns().add(numberColumn);

        /* Finish making the GUI, with "Add" and "Delete"
buttons placed into
         * the window below the table. */

        TextField nameInput = new TextField();
        nameInput.setPrefColumnCount(10);
        nameInput.setPromptText("(name)");
        TextField numberInput = new TextField();
        numberInput.setPromptText("(number)");
        numberInput.setPrefColumnCount(10);

        Button deleteButton = new Button("Delete Selected
Entry");
        deleteButton.setOnAction( e -> {
            int selected =
phoneBook.getSelectionModel().getSelectedIndex();
            if (selected >= 0)
                phoneEntries.remove(
phoneBook.getSelectionModel().getSelectedIndex());
        });
        deleteButton.setMaxWidth(Double.POSITIVE_INFINITY);
        deleteButton.disableProperty().bind(

phoneBook.getSelectionModel().selectedIndexProperty().isEqu
alTo(-1));
        Button addButton = new Button("Add:");
        addButton.setOnAction( e -> {  // add a new row to
the table
            String name = nameInput.getText().trim();
            String number = numberInput.getText().trim();
            if (name.length() == 0 ) {
```

```
                    // Don't allow empty name in the table
                error("You must enter a name and number
before adding an entry.");
                nameInput.requestFocus();
                return;
            }
            if (number.length() == 0 ) {
                    // Don't allow an empty phone number in
the table
                error("You must enter a name and number
before adding an entry.");
                numberInput.requestFocus();
                return;
            }
            nameInput.setText("");   // empty the input
boxes, since the data has been
            numberInput.setText(""); //        copied into
the table
            PhoneEntry newEntry = new
PhoneEntry(name,number);
            phoneEntries.add( newEntry );
            phoneBook.scrollTo(phoneEntries.size() - 1); //
make sure new row is visib.e

phoneBook.getSelectionModel().select(phoneEntries.size() -
1); // highlight new entry
        });
        HBox add = new
HBox(8,addButton,nameInput,numberInput);
        add.setPadding( new Insets(5) );
        VBox buttons = new VBox(add,deleteButton);
        buttons.setStyle("-fx-border-color:black; -fx-
border-width: 2px");

        BorderPane tableHolder = new BorderPane(phoneBook);
        tableHolder.setBottom(buttons);

        stage.setOnHidden( e -> savePhoneBook(dataFile,
phoneEntries) );

        stage.setScene( new Scene(tableHolder) );
        stage.setTitle("Phone Book Editor");
        stage.show();

    } // end start();


    /**
```

```
     * If the data file already exists, load the entries
from the data
     * file in the list (which is the items list from the
TableView).
     * The format of the file must be as follows:  Each
line of the file
     * represents one directory entry, with the name and
the number for that
     * entry separated by the character '%'.  If a file
exists but does not
     * have this format, then the program terminates; this
is done to
     * avoid overwriting a file that is being used for
another purpose.
     */
    private void loadPhoneBook(File dataFile,
List<PhoneEntry> entries) {
        if ( ! dataFile.exists() ) {
            message("No phone book data file found.  A new
one\n"
                        + "will be created when the program
ends,\n"
                        + "if you add any entries to the
table.\n"
                        + "File name:\n    " +
dataFile.getAbsolutePath());
        }
        else {
            try( Scanner scanner = new Scanner(dataFile) )
{
                while (scanner.hasNextLine()) {
                    String phoneEntry = scanner.nextLine();
                    int separatorPosition =
phoneEntry.indexOf('%');
                    if (separatorPosition == -1)
                        throw new IOException("File is not
a phonebook data file.");
                    String name = phoneEntry.substring(0,
separatorPosition);
                    String number =
phoneEntry.substring(separatorPosition+1);
                    entries.add( new
PhoneEntry(name,number) );
                }
            }
            catch (IOException e) {
                error("Error in phone book data file.\n"
                            + "This program cannot
```

```
continue.\n"
                            + "Data File name: \n   " +
dataFile.getAbsolutePath());
                System.exit(1);
            }
        }
    }


    /**
     * Write the phone book entries from the list to the
data files.
     * But if the list is empty, don't write anything.
     */
    private void savePhoneBook(File dataFile,
List<PhoneEntry> entries) {
        if (entries.size() == 0)
            return;
        System.out.println("Saving phone directory to file
" +
                dataFile.getAbsolutePath() + " ...");
        PrintWriter out;
        try {
            out = new PrintWriter( dataFile );
        }
        catch (IOException e) {
            error("ERROR: Can't open data file for output."
                    + "Phone book data cannot be saved.");
            return;
        }
        for ( PhoneEntry entry : entries ) {
            out.println( entry.nameProperty().get() + "%" +
entry.numberProperty().get() );
        }
        out.flush();
        out.close();
        if (out.checkError())
            error("ERROR: Some error occurred while writing
the data file."
                    + "Phone book data might not have been
saved correctly.");
    }


    /**
     *  A utility method for showing an informational
message to the user.
     */
```

```
    private void message(String text) {
        Alert alert = new
Alert(Alert.AlertType.INFORMATION, text);
        alert.setHeaderText(null);
        alert.showAndWait();
    }

    /**
     *  A utility method for showing an erro message to the
user.
     */
    private void error(String text) {
        Alert alert = new Alert(Alert.AlertType.ERROR,
text);
        alert.setHeaderText(null);
        alert.showAndWait();
    }


} // end PhoneDirectoryGUI
```