

Introduction to Programming Using Java

Version 4.0, July 2002

Author: David J. Eck

**Department of Mathematics and Computer Science
Hobart and William Smith Colleges
Geneva, New York 14456
Email: eck@hws.edu
WWW: <http://math.hws.edu/eck/>**

**This PDF file or printout contains parts of a free textbook
that covers introductory programming with Java.
The entire text is available on the World-Wide Web,
for use on-line and for downloading,
at this Web address:**

<http://math.hws.edu/javanotes/>

The PDF file and printouts that are made from it do not show Java applets that are embedded throughout the text. Also not included are Java source code examples from Appendix 3 and solutions to the quizzes and programming exercises. This version of the textbook requires Java 1.3 or higher.

© 2002, David J. Eck. This is a free textbook. There are no restrictions on using or redistributing a complete, unmodified copy of this material. There are some restrictions on modified copies. To be precise: Permission is granted to copy, distribute and/or modify this document under the terms of the [GNU Free Documentation License](http://www.gnu.org/licenses/old/licenses.html), Version 1.1 or any later version published by the Free Software Foundation; with no invariant sections, front cover text, or back cover text.

Introduction to Programming Using Java

Version 4.0, July 2002

Requires Java 1.3 or higher

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

WELCOME TO *Introduction to Programming Using Java*, the fourth edition of a free, on-line textbook on introductory programming, which uses Java as the language of instruction. Previous versions have been used as a textbook for an introductory programming class at Hobart and William Smith Colleges. See <http://math.hws.edu/eck/cs124/> for information about this course. This on-line book contains Java applets, many of which require Java 1.3 or higher. To see these applets, you will need a Web browser that uses a recent version of Java. To learn more, please read the [preface](#).

Links for downloading copies of this text can be found at the bottom of this page.

Search this Text: Although this book does not have a conventional index, you can search it for terms that interest you. Note that this feature searches the book at its on-line site, so you must be working on-line to use it.

Search *Introduction to Programming Using Java* for pages...

Short Table of Contents:

- [Full Table of Contents](#)
- [Preface](#)
- Chapter 1: [Overview: The Mental Landscape](#)
- Chapter 2: [Programming in the Small I: Names and Things](#)
- Chapter 3: [Programming in the Small II: Control](#)
- Chapter 4: [Programming in the Large I: Subroutines](#)
- Chapter 5: [Programming in the Large II: Objects and Classes](#)
- Chapter 6: [Applets, HTML, and GUI's](#)
- Chapter 7: [Advanced GUI Programming](#)
- Chapter 8: [Arrays](#)
- Chapter 9: [Correctness and Robustness](#)
- Chapter 10: [Advanced Input/Output](#)
- Chapter 11: [Linked Data Structures and Recursion](#)

- Chapter 12: [Generic Programming and Collection Classes](#)
- Appendix 1: [Other Features of Java](#)
- Appendix 2: [Some Notes on Java Programming Environments](#)
- Appendix 3: [Source Code for All Examples in this Book](#)
- [News and Errata](#)

© 2002, David J. Eck. This is a free textbook. There are no restrictions on using or redistributing or posting on the web a complete, unmodified copy of this material. There are some restrictions on modified copies. To be precise: Permission is granted to copy, distribute and/or modify this document under the terms of the [GNU Free Documentation License](#), Version 1.1 or any later version published by the Free Software Foundation; with no invariant sections, front cover text, or back cover text.

The most recent version of this book is always available, at no charge, for downloading and for on-line use at the Web address <http://math.hws.edu/javanotes/>. The previous edition, which covered Java 1.1, can be found at <http://math.hws.edu/eck/cs124/javanotes3/>.

Downloading Links

Use one of the following links to download a compressed archive of this textbook:

- Windows: <http://math.hws.edu/eck/cs124/downloads/javanotes4.zip> (1.8 Megabytes), with text files in Windows/DOS format. This archive can be extracted with [WinZip](#), or with the free program, [Aladdin](#) Stuffit Expander for Windows, available from <http://www.stuffit.com/expander/>.
- Linux/UNIX and MacOS X: <http://math.hws.edu/eck/cs124/downloads/javanotes4.tar.bz2> (1.0 Megabytes), with text files in Linux/UNIX format. If you have the bzip2 program, you should be able to extract this archive with the commands "bunzip2 javanotes4.tar.bz2" followed by "tar xf javanotes4.tar". On Macintosh, this archive can be extracted using [Aladdin](#) Stuffit Expander for Macintosh, available from <http://www.stuffit.com/expander/>.
- Linux/UNIX: <http://math.hws.edu/eck/cs124/downloads/javanotes4.tar.Z> (2.0 Megabytes), with text files in Linux/UNIX format. If you can't use the previous archive, try this one. You can extract this archive on most UNIX systems with the commands "uncompress javanotes4.tar.Z" followed by "tar xf javanotes4.tar".

I know from experience that a lot of people will want to print all or part of the text. The following PDF file is provided to make this a little easier. This is nothing fancy -- just the Web pages captured in a single file. To use this file, you need Adobe Acrobat Reader Version 4 or later. (When you click on this link, the file might open in your Web browser; to download it, right-click the link and choose "Save Link As" or similar command.)

- <http://math.hws.edu/eck/cs124/downloads/javanotes4.pdf> (2.1 Megabytes; 554 pages)

[David Eck](#) (eck@hws.edu)

Version 4.0, July 2002

Introduction to Programming Using Java, Fourth Edition

Table of Contents

THIS IS THE FULL TABLE OF CONTENTS for version 4.0 of an on-line introductory programming textbook. For more information about the text, please see its [front page](#). The text is available on-line at <http://math.hws.edu/javanotes/>.

[Preface](#)

Chapter 1: [Overview: The Mental Landscape](#)

- 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 World-Wide Web](#)
- [Quiz on this Chapter](#)

Chapter 2: [Programming in the Small I: Names and Things](#)

- Section 1: [The Basic Java Application](#)
- Section 2: [Variables and the Primitive Types](#)
- Section 3: [Strings, Objects, and Subroutines](#)
- Section 4: [Text Input and Output](#)
- Section 5: [Details of Expressions](#)
- [Programming Exercises](#)
- [Quiz on this Chapter](#)

Chapter 3: [Programming in the Small II: Control](#)

- 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 Applets and Graphics](#)
- [Programming Exercises](#)
- [Quiz on this Chapter](#)

Chapter 4: [Programming in the Large I: Subroutines](#)

- [Section 1: Black Boxes](#)
- [Section 2: Static Subroutines and Static Variables](#)
- [Section 3: Parameters](#)
- [Section 4: Return Values](#)
- [Section 5: Toolboxes, API's, and Packages](#)
- [Section 6: More on Program Design](#)
- [Section 7: The Truth about Declarations](#)
- [Programming Exercises](#)
- [Quiz on this Chapter](#)

Chapter 5: [Programming in the Large II: Objects and Classes](#)

- [Section 1: Objects, Instance Variables, and Instance Methods](#)
- [Section 2: Constructors and Object Initialization](#)
- [Section 3: Programming with Objects](#)
- [Section 4: Inheritance, Polymorphism, and Abstract Classes](#)
- [Section 5: this and super](#)
- [Section 6: Interfaces, Nested Classes and Other Details](#)
- [Programming Exercises](#)
- [Quiz on this Chapter](#)

Chapter 6: [Applets, HTML, and GUI's](#)

- [Section 1: The Basic Java Applet and JApplet](#)
- [Section 2: HTML Basics and the Web](#)
- [Section 3: Graphics and Painting](#)
- [Section 4: Mouse Events](#)
- [Section 5: Keyboard Events](#)
- [Section 6: Introduction to Layouts and Components](#)
- [Programming Exercises](#)
- [Quiz on this Chapter](#)

Chapter 7: [Advanced GUI Programming](#)

- [Section 1: More about Graphics](#)
- [Section 2: More about Layouts and Components](#)
- [Section 3: Basic Components and Their Events](#)

- Section 4: [Programming with Components](#)
- Section 5: [Menus and Menubars](#)
- Section 6: [Timers, Animation, and Threads](#)
- Section 7: [Frames and Applications](#)
- [Programming Exercises](#)
- [Quiz on this Chapter](#)

Chapter 8: [Arrays](#)

- Section 1: [Creating and Using Arrays](#)
- Section 2: [Programming with Arrays](#)
- Section 3: [Dynamic Arrays, ArrayLists, and Vectors](#)
- Section 4: [Searching and Sorting](#)
- Section 5: [Multi-Dimensional Arrays](#)
- [Programming Exercises](#)
- [Quiz on this Chapter](#)

Chapter 9: [Correctness and Robustness](#)

- Section 1: [Introduction to Correctness and Robustness](#)
- Section 2: [Writing Correct Programs](#)
- Section 3: [Exceptions and the `try...catch` Statement](#)
- Section 4: [Programming with Exceptions](#)
- [Programming Exercises](#)
- [Quiz on this Chapter](#)

Chapter 10: [Advanced Input/Output](#)

- Section 1: [Streams, Readers, and Writers](#)
- Section 2: [Files](#)
- Section 3: [Programming with Files](#)
- Section 4: [Networking](#)
- Section 5: [Programming Networked Applications](#)
- [Programming Exercises](#)
- [Quiz on this Chapter](#)

Chapter 11: [Linked Data Structures and Recursion](#)

- Section 1: [Recursion](#)
- Section 2: [Linking Objects](#)
- Section 3: [Stacks and Queues](#)
- Section 4: [Binary Trees](#)
- Section 5: [A Simple Recursive-descent Parser](#)

- [Programming Exercises](#)
- [Quiz on this Chapter](#)

Chapter 12: [Generic Programming and Collection Classes](#)

- Section 1: [Generic Programming](#)
- Section 2: [List and Set Classes](#)
- Section 3: [Map Classes](#)
- Section 4: [Programming with Collection Classes](#)
- [Programming Exercises](#)
- [Quiz on this Chapter](#)

Appendix 1: [Other Features of Java](#)

Appendix 2: [Some Notes on Java Programming Environments](#)

Appendix 3: [Source code for all examples in the text](#)

[News and Errata](#)

[David J. Eck](#) (eck@hws.edu), July 2002

Chapter 1

Overview: The Mental Landscape

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

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

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 World-Wide Web](#)
- [Quiz on this Chapter](#)

[[First Section](#) | [Next Chapter](#) | [Main Index](#)]

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 it can directly execute a program only if it is expressed in that language. (It can execute programs written in other languages if they are first translated into machine language.)

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

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

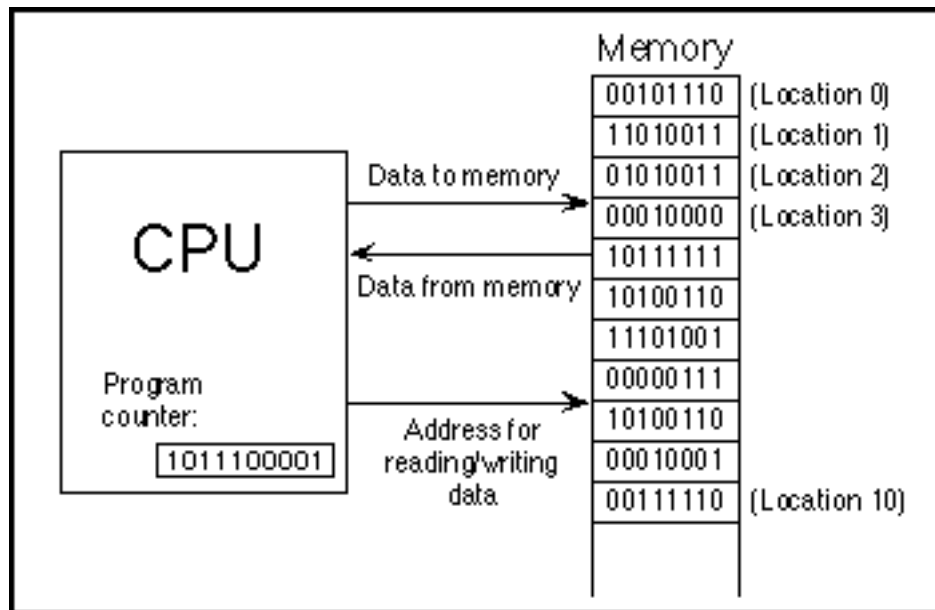
The details of the fetch-and-execute cycle are not terribly important, but there are a few basic things you should know. The CPU contains a few internal **registers**, which are small memory units capable of holding a single number or machine language instruction. The CPU uses one of these registers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

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

Machine language instructions are expressed as binary numbers. A binary number is made up of just two possible digits, zero and one. So, a machine language instruction is just a sequence of zeros and ones. Each particular sequence encodes some particular instruction. The data that the computer manipulates is also encoded as binary numbers. A computer can work directly with binary numbers because switches can readily represent such numbers: Turn the switch on to represent a one; turn it off to represent a zero.

Machine language instructions are stored in memory as patterns of switches turned on or off. When a machine language instruction is loaded into the CPU, all that happens is that certain switches are turned on or off in the pattern that encodes that particular instruction. The CPU is built to respond to this pattern by executing the instruction it encodes; it does this simply because of the way all the other switches in the CPU are wired together.

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



This figure is taken from [The Most Complex Machine: A Survey of Computers and Computing](#), a textbook that serves as an introductory overview of the whole field of computer science. If you would like to know more about the basic operation of computers, please see Chapters 1 to 3 of that text.

[[Next Section](#) | [Chapter Index](#) | [Main Index](#)]

Section 1.2

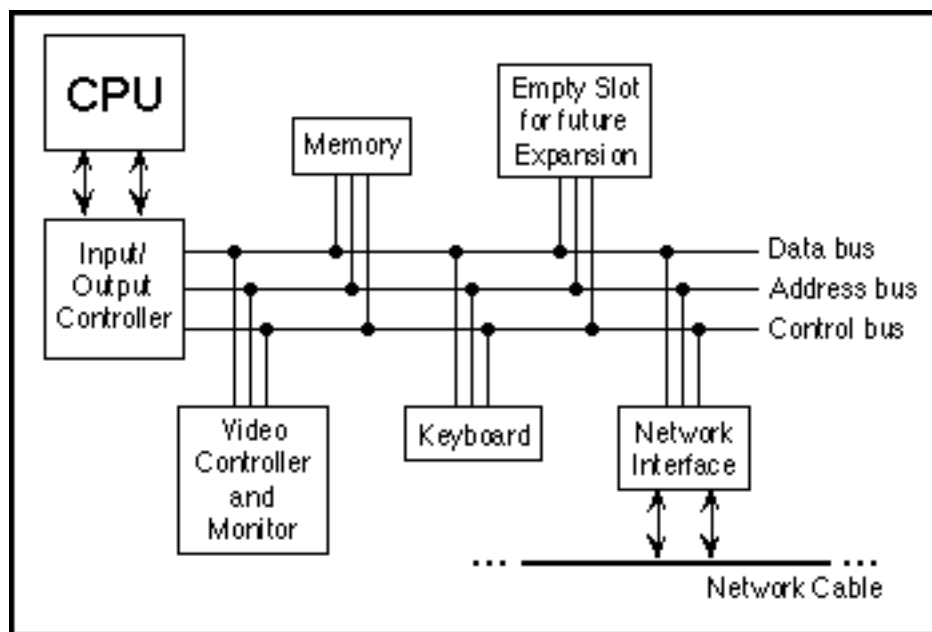
Asynchronous Events: Polling Loops and Interrupts

THE CPU SPENDS ALMOST ALL ITS TIME fetching instructions from memory and executing them. However, the CPU and main memory are only two out of many components in a real computer system. A complete system contains other devices such as:

- A **hard disk** for storing programs and data files. (Note that main memory holds only a comparatively small amount of information, and holds it only as long as the power is turned on. A hard disk is necessary for permanent storage of larger amounts of information, but programs have to be loaded from disk into main memory before they can actually be executed.)
- A **keyboard** and **mouse** for user input.
- A **monitor** and **printer** which can be used to display the computer's output.
- A **modem** that allows the computer to communicate with other computers over telephone lines.
- A **network interface** that allows the computer to communicate with other computers that are connected to it on a network.
- 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:



(This illustration is taken from [The Most Complex Machine](#).)

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

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

Again, you should understand that this is 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, it saves enough information about what it is currently doing so that it can return to the same state later. This information consists of the contents of important internal registers such as the program counter. Then the CPU jumps to some predetermined memory location and begins executing the instructions stored there. Those instructions make up an **interrupt handler** that does the processing necessary to respond to the interrupt. (This interrupt handler is part of the device driver software for the device that signalled the interrupt.) At the end of the interrupt handler is an instruction that tells the CPU to jump back to what it was doing; it does that by restoring its previously saved state.

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

As another example of how interrupts are used, consider what happens when the CPU needs to access data that is stored on the hard disk. The CPU can only access data directly 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 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 even modern personal computers with a single user use 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.) At any given time, only one thread can actually be executed by a CPU. The CPU will continue running the same thread 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 have a chance to run. When the event occurs, an interrupt will "wake up" the thread so that it can continue running.
- The thread might use up its allotted slice of time and be suspended to allow other threads to run. Not all computers can "forcibly" suspend a thread in this way; those that can are said to use **preemptive multitasking**. To do preemptive multitasking, a computer needs a special timer device that generates an interrupt at regular intervals, such as 100 times per second. When a timer interrupt occurs, the CPU has a chance to switch from one thread to another, whether the thread that is currently running likes it or not.

Ordinary users, and indeed ordinary programmers, have no need to deal with interrupts and interrupt handlers. They can concentrate on the different tasks or threads that they want the computer to perform; the details of how the computer manages to get all those tasks done are not important to them. In fact, most users, and many programmers, can ignore threads and multitasking altogether. However, threads have become increasingly important as computers have become more powerful and as they have begun to make more use of multitasking. Indeed, threads are built into the Java programming language as a fundamental programming concept.

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 specified 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.

By the way, the software that does all the interrupt handling and the communication with the user and with hardware devices is called the **operating system**. The operating system is the basic, essential software without which a computer would not be able to function. Other programs, such as word processors and World Wide Web browsers, are dependent upon the operating system. Common operating systems include UNIX, Linux, DOS, Windows 98, Windows 2000 and the Macintosh OS.

[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

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, Pascal, or C++. A program written in a high-level language cannot be run directly on any computer. First, it has to be translated into machine language. This translation can be done by a program called a **compiler**. A compiler takes a high-level-language program and translates it into an executable machine-language program. Once the translation is done, the machine-language program can be run any number of times, but of course it can only be run on one type of computer (since each type of computer has its own individual machine language). If the program is to run on another type of computer it has to be re-translated, using a different compiler, into the appropriate machine language.

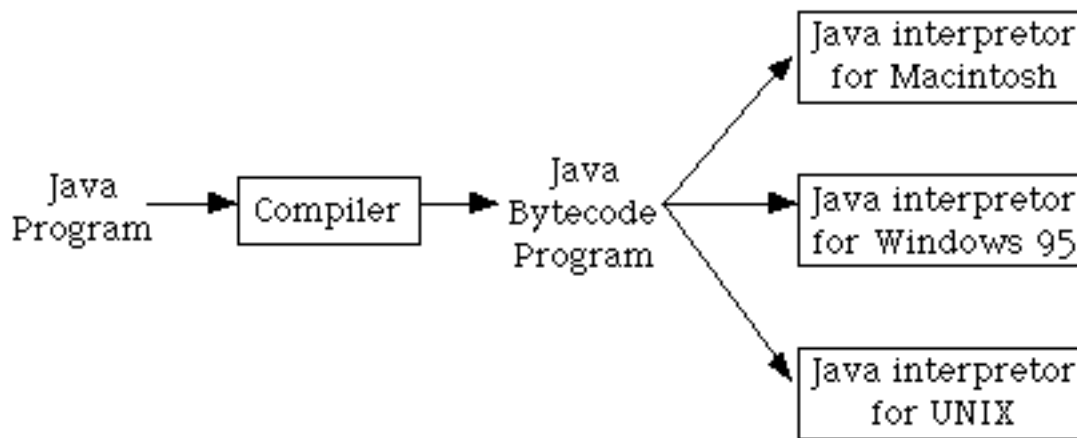
There is an alternative to compiling a high-level language program. Instead of using a compiler, which translates the program all at once, you can use an **interpreter**, which translates it instruction-by-instruction, as necessary. An interpreter is a program that acts much like a CPU, with a kind of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

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

The designers of Java chose to use a combination of compilation and interpretation. Programs written in Java are compiled into machine language, but it is a machine language for a computer that doesn't really exist. This so-called "virtual" computer is known as the **Java virtual machine**. The machine language for the Java virtual machine is called **Java bytecode**. There is no reason why Java bytecode could not be used as the machine language of a real computer, rather than a virtual computer. In fact, Sun Microsystems -- the originators of Java -- have developed CPU's that run Java bytecode as their machine language.

However, one of the main selling points of Java is 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 Java virtual machine in the same way that Virtual PC simulates a PC computer.**

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



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

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

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

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 6](#).

[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

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 locations treated as a unit) that has been given a name so that it can be easily referred to and used in a program. The programmer only has to worry about the name; it is the compiler's responsibility to keep track of the memory location. The programmer does need to keep in mind that the name refers to a kind of "box" in memory that can hold data, even if the programmer doesn't have to know where in memory that box is located.

In Java and most other 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 type of data that a program might need to store.

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

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

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

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

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

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

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

(Don't worry about the details for now. Just remember that the computer can test a condition and decide

what to do next on the basis of that test.)

Loops are used when the same task has to be performed more than once. For example, if you want to print out a mailing label for each name on a mailing list, you might say, "Get the first name and address and print the label; get the second name and address and print the label; get the third name and address and print the label..." But this quickly becomes ridiculous -- and might not work at all if you don't know in advance how many names there are. What you would like to say is something like "While there are more names to process, get the next name and address, and print the label." A loop can be used in a program to express such repetition.

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

```
drawHouse ( ) ;
```

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

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

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

[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

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 tends 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 as 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

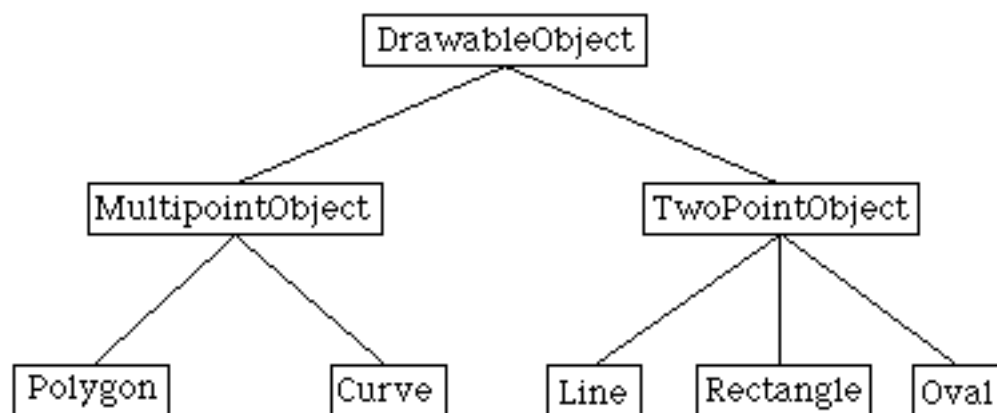
data and subroutines. The point-of-view in OOP is that an object is a kind of self-sufficient entity that has an internal **state** (the data it contains) and that can respond to **messages** (calls to its subroutines). A mailing list object, for example, has a state consisting of a list of names and 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 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 relationship" to one another. Objects that contain the same type of data and that respond to the same messages in the same way belong to the same **class**. (In actual programming, the class is primary; that is, a class is created and then one or more objects are created using that class as a template.) But objects can be similar without being in exactly the same class.

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



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

Inheritance is a powerful means for organizing a program. It is also related to the problem of reusing

software components. A class is the ultimate reusable component. Not only can it be reused directly if it fits exactly into a program you are trying to write, but if it just almost fits, you can still reuse it by defining a subclass and making only the small changes necessary to adapt it exactly to your needs.

So, OOP is meant to be both a superior program-development tool and a partial solution to the software reuse problem. Objects, classes, and object-oriented programming will be important themes throughout the rest of this text.

[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

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. Assuming that you are reading these notes on a computer, 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 Macintosh, Windows, and various UNIX window systems. 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 in appearance from platform to platform, but their functionality should be identical on any computer on which the program runs.

Below is a very simple Java program -- actually an "**applet**," since it is running right here in the middle of a page -- that shows a few standard GUI interface components. There are four components that you can interact with: a button, a checkbox, a text field, and a pop-up menu. These components are labeled. There are a few other components in the applet. The labels themselves are components (even though you can't interact with them). The right half of the applet is a text area component, which can display multiple lines of text. In fact, in Java terminology, the whole applet is itself considered to be a "component." Try clicking on the button and on the checkbox, and try selecting an item from the pop-up menu. You will see a message in the text area about each action that you perform. You can type in the text field, but you might have to click on it first to activate it. When you press return while typing in the text field, you will see a message in the text area:

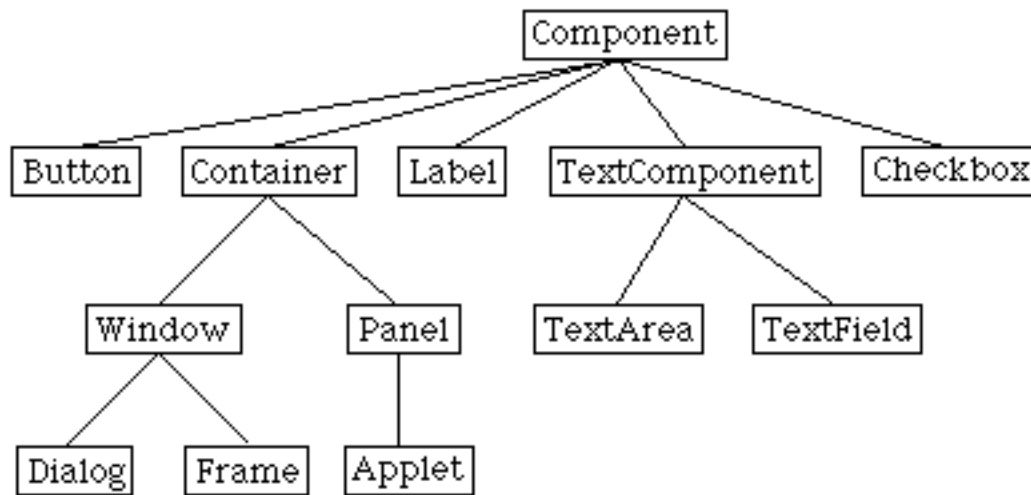
(Applet "GUIDemo" would be displayed here
if Java were available.)

Now, Java actually has two complete sets of GUI components. One of these, the AWT or **Abstract Windowing Toolkit**, was available in the original version of Java. The other, which is known as **Swing**, is included in Java version 1.2 or later. Here is a version of the applet that uses Swing instead of the AWT. If you see the above applet but just see a blank area here, it means that you are using a Web browser that uses an older version of Java:

(Applet "GUIDemo2" would be displayed here
if Java were available.)

As you interact with the GUI components in these applets, you generate "events." For example, clicking a push button generates an event. Each time an event is generated, a message is sent to the applet telling it that the event has occurred, and the applet responds according to its program. In fact, the program consists mainly of "event handlers" that tell the applet how to respond to various types of events. In this example, the applets have been programmed to respond to each event by displaying a message in the text area.

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



Don't worry about the details for now, but try to get some feel about how object-oriented programming and inheritance are used here. Note that all the GUI classes are subclasses, directly or indirectly, of a class called `Component`. Two of the direct subclasses of `Component` themselves have subclasses. The classes `TextArea` and `TextField`, which have certain behaviors in common, are grouped together as subclasses of `TextComponent`. The class named `Container` refers to components that can contain other components. The `Applet` class is, indirectly, a subclass of `Container` since applets can contain components such as buttons and text fields.

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

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

Section 1.7

The Internet and the World-Wide Web

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

Today, millions of computers throughout the world are connected to a single huge network called the **Internet**. New computers are being connected to the Internet every day. In fact, a computer can join the Internet temporarily by using a modem to establish a connection through telephone lines.

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 files and electronic mail.

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

Every computer on the Internet has an **IP address**, a number that identifies it uniquely among all the computers on the net. The IP address is used for addressing packets. A computer can only send data to another computer on the Internet if it knows that computer's IP address. Since people prefer to use names rather than numbers, many computers are also identified by names, called **domain names**. For example, the main computer at Hobart and William Smith Colleges has the domain name hws3.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 remote login, electronic mail, FTP, and the World-Wide Web.

Remote login allows a person using one computer to log on to another computer. (Of course, that person needs to know a user name and password for an account on the other computer.) There are several different protocols for remote login, including the traditional **telnet** and the more secure **ssh** (secure shell). Telnet and ssh provide only a command-line interface. Essentially, the first computer acts as a terminal for the second. Remote login is often used by people who are away from home to access their computer accounts back home -- and they can do so from any computer on the Internet, anywhere in the world.

Electronic mail, or email, provides person-to-person communication over the Internet. An email message is sent by a particular user of one computer to a particular user of another computer. Each person is identified by a unique email address, which consists of the domain name of the computer where they receive their mail together with their user name or personal name. The email address has the form

"username@domain.name". For example, my own email address is: eck@hws.edu. Email is actually transferred from one computer to another using a protocol called **SMTP** (Simple Mail Transfer Protocol). Email might still be the most common and important use of the Internet, although it has certainly been challenged in popularity by the World-Wide Web.

FTP (File Transport Protocol) is designed to copy files from one computer to another. As with remote login, an FTP user needs a user name and password to get access to a computer. However, many computers have been set up with special accounts that can be accessed through FTP with the user name "anonymous" and any password. This so-called **anonymous FTP** can be used to make files on one computer publically available to anyone with Internet access.

The **World-Wide Web** (WWW) is based on **pages** which can contain information of many different kinds as well as **links** to other pages. These pages are viewed with a **Web browser** program such as Netscape or Internet Explorer. Many people seem to think that the World-Wide Web is the Internet, but it's really just a graphical user interface to the Internet. The pages that you view with a Web browser are just files that are stored on computers connected to the Internet. When you tell your Web browser to load a page, it contacts the computer on which the page is stored and transfers it to your computer using a protocol known as **HTTP** (HyperText Transfer Protocol). Any computer on the Internet can publish pages on the World-Wide Web. When you use a Web browser, you have access to a huge sea of interlinked information that can be navigated with no special computer expertise. The Web is the most exciting part of the Internet and is driving the Internet to a truly phenomenal rate of growth. If it fulfills its promise, the Web might become a universal and fundamental part of everyday life.

I should note that a typical Web browser can use other protocols besides HTTP. For example, it can also use FTP to transfer files. The traditional user interface for FTP was a command-line interface, so among all the other things it does, a Web browser provides a modern graphical user interface for FTP. This allows people to use FTP without even knowing that there is such a thing! (This fact should help you understand the difference between a program and a protocol. FTP is not a program. It is a protocol, that is, a set of standards for a certain type of communication between computers. To use FTP, you need a program that implements those standards. Different FTP programs can present you with very different user interfaces. Similarly, different Web browser programs can present very different interfaces to the user, but they must all use HTTP to get information from the Web.)

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

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

End of Chapter 1

[[Next Chapter](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Quiz Questions For Chapter 1

THIS PAGE CONTAINS A SAMPLE quiz on material from [Chapter 1](#) of this [on-line Java textbook](#). You should be able to answer these questions after studying that chapter. Sample answers to all the quiz questions can be found [here](#).

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?)

[[Answers](#) | [Chapter Index](#) | [Main Index](#)]

Chapter 2

Programming in the Small I Names and Things

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

Programming in the small, which is sometimes called **coding**, would then refer to filling in the details of that design. The details are the explicit, step-by-step instructions for performing fairly small-scale tasks. When you do coding, you are working fairly "close to the machine," with some of the same concepts that you might use in machine language: memory locations, arithmetic operations, loops and decisions. 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.

Contents of Chapter 2:

- Section 1: [The Basic Java Application](#)
 - Section 2: [Variables and the Primitive Types](#)
 - Section 3: [Strings, Objects, and Subroutines](#)
 - Section 4: [Text Input and Output](#)
 - Section 5: [Details of Expressions](#)
 - [Programming Exercises](#)
 - [Quiz on this Chapter](#)
-

[[First Section](#) | [Next Chapter](#) | [Previous Chapter](#) | [Main Index](#)]

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**. A semantically correct program is one that does what you want it to.

When I introduce a new language feature in these notes, I will explain both the syntax and the semantics of that feature. You should memorize the syntax; that's the easy part. Then you should try to get a feeling for the semantics by following the examples given, making sure that you understand how they work, and maybe writing short programs of your own to test your understanding.

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 can't tell you 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 [Appendix 2](#) for information on some common Java 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, 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:

```
public class HelloWorld {
```

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

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

} // end of class HelloWorld
```

The command that actually displays the message is:

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

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

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

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

Everything else in the program is required by the rules of Java syntax. All programming in Java is done inside "classes." The first line in the above program 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 called `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 the `main()` subroutine, and the statements that it contains are executed. These statements make up the script that tells the computer exactly what to do when the program is executed. The `main()` routine can call subroutines that are defined in the same class or even in other classes, but it is the `main()` routine that determines how and in what order the other subroutines are used.

The word "public" in the first line of `main()` means that this routine can be called from outside the program. This is essential because the `main()` routine is called by the Java interpreter. 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** (which is **green** if your browser supports colored text) is a

placeholder that describes something you need to type when you write an actual program.

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

```
public class program-name {

    optional-variable-declarations-and-subroutines

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

    optional-variable-declarations-and-subroutines

}
```

The name on the first line is the name of the program, as well as the name of the class. If the name of the class is HelloWorld, then the class should 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 Java bytecode that is 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.

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 (starting with variables, in the next section).

By the way, recall that one of the neat features of Java is that it can be used to write applets that can run on pages in a Web browser. Applets are very different things from stand-alone programs such as the HelloWorld program, and they are not written in the same way. For one thing, an applet doesn't have a main() routine. Applets will be covered in [Chapter 6](#) and [Chapter 7](#). In the meantime, you will see applets in this text that simulate stand-alone programs. The applets you see are not really the same as the stand-alone programs that they simulate, since they run right on a Web page, but they will have the same behavior as the programs I describe. Here, just for fun, is an applet simulating the HelloWorld program. To run the program, click on the button:

(Applet "ConsoleApplet" would be displayed here
if Java were available.)

[[Next Section](#) | [Previous Chapter](#) | [Chapter Index](#) | [Main Index](#)]

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

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

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

Uppercase and lowercase letters are considered to be different, so that HelloWorld, helloworld, HELLOWORLD, and hElLoWoRlD are all distinct names. Certain names are reserved for special uses in Java, and cannot be used by the programmer for other purposes. These **reserved words** include: class, public, static, if, else, while, and several dozen other words.

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

Finally, I'll note that often things are referred to by "compound names" which consist of several ordinary names separated by periods. 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". I'll use the term **identifier** to refer to any name -- single or compound -- that can be used to refer to something in Java. (Note that the reserved words are **not identifiers**, since they can't be used as names for things.)

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

Variables are actually rather subtle. Properly speaking, a variable is not a name for the data itself but for a location in memory that can hold data. You should think of a variable as a container or box where you can store data that you will need to use later. The variable refers directly to the box and only indirectly to the data in the box. Since the data in the box can change, a variable can refer to different data values at different times during the execution of the program, but it always refers to the same box. Confusion can arise, especially for beginning programmers, because when a variable is used in a program in certain ways, it refers to the container, but when it is used in other ways, it refers to the data in the container. You'll see examples of both cases below.

(In this way, a variable is something like the title, "The President of the United States." This title can refer to different people at different time, but it always refers to the same office. If I say "the President went fishing," I mean that George W. Bush went fishing. But if I say "Hillary Clinton wants to be President" I mean that she wants to fill the office, not that he wants to be George Bush.)

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`.

(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 "`=`".)

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

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

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

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

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

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

A variable of type `char` occupies two bytes in memory. The value of a `char` variable is a single character such as A, *, x, or a space character. The value can also be a special character such a tab or a carriage return or one of the many Unicode characters that come from different languages. When a character is typed into a program, it must be

surrounded by single quotes; for example: 'A', '*', or 'x'. Without the quotes, A would be an identifier and * would be a multiplication operator. The quotes are not part of the value and are not stored in the variable; they are just a convention for naming a particular character constant in a program.

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

Numeric literals are a little more complicated than you might expect. Of course, there are the obvious literals such as 317 and 17.42. But there are other possibilities for expressing numbers in a Java program. First of all, real numbers can be represented in an exponential form such as 1.3e12 or 12.3737e-108. The "e12" and "e-108" represent powers of 10, so that 1.3e12 means 1.3 times 10¹² and 12.3737e-108 means 12.3737 times 10⁻¹⁰⁸. This format is used for very large and very small numbers. Any numerical literal that contains a decimal point or exponential is a literal of type `double`. To make a literal of type `float`, you have to append an "F" or "f" to the end of the number. For example, "1.2F" stands for 1.2 considered as a value of type `float`. (Occasionally, you need to know this because the rules of Java say that you can't assign a value of type `double` to a variable of type `float`, so you might be confronted with a ridiculous-seeming error message if you try to do something like `float x = 1.2;`. You have to say `float x = 1.2F;`. This is one reason why I advise sticking to type `double` for real numbers.)

Even for integer literals, there are some complications. Ordinary integers such as 177777 and -32 are literals of type `byte`, `short`, or `int`, depending on their size. You can make a literal of type `long` by adding "L" as a suffix. For example: 17L or 728476874368L. As another complication, Java allows octal (base-8) and hexadecimal (base-16) literals. (I don't want to cover base-8 and base-16 in these notes, but in case you run into them in other people's programs, it's worth knowing that a zero at the beginning of an integer makes it an octal literal, as in 045 or 077. A hexadecimal literal begins with 0x or 0X, as in 0x45 or 0xFF7A. By the way, the octal literal 045 represents the number 37, not the number 45.)

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

```
rate > 0.05
```

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

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

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

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

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

Because strings are objects, their behavior in programs is peculiar in some respects (to someone who is not used to objects). I'll have more to say about them in the [next section](#).

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;
```

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

```
public class Interest {

    /*
     * This class implements a simple program that
     * will compute the amount of interest that is
     * earned on $17,000 invested at an interest
     * rate of 0.07 for one year. The interest and
     * the value of the investment after one year are
     * printed to standard output.
     */

    public static void main(String[] args) {

        /* Declare the variables. */

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

        /* Do the computations. */

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

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

        /* Output the results. */

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

    } // end of main()
}
```

```
} // end of class Interest
```

And here is an applet that simulates this program:

(Applet "Interest1Console" would be displayed here
if Java were available.)

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 carriage return after the end of the information that it displays, while `System.out.print` does not. Thus, the value of `interest`, which is displayed by the subroutine call `"System.out.println(interest);"`, follows on the same line after the string displayed by the previous 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.

[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Section 2.3

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

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

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

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

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

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

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

(As an aside, I will note that the object referred to by `System.out` is an object of the class `PrintStream`. `PrintStream` is another class that is a standard part of Java. Any object of type `PrintStream` is a destination to which information can be printed; any object of type `PrintStream` has a `print` subroutine that can be used to send information to that destination. The object `System.out` is just one possible destination, and `System.out.print` is the subroutine that sends information to that destination. Other objects of type `PrintStream` might send information to other destinations such as files or across a network to

other computers. This is object-oriented programming: Many different things which have something in common -- they can all be used as destinations for information -- can all be used in the same way -- through a `print` subroutine. The `PrintStream` class expresses the commonalities among all these objects.)

Since class names and variable names are used in similar ways, it might be hard to tell which is which. 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 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.

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

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

You are familiar with the mathematical function that computes the square root of a number. Java has a corresponding function called `Math.sqrt`. This function is a static member subroutine of the class named `Math`. If `x` is any numerical value, then `Math.sqrt(x)` computes and returns the square root of that value. Since `Math.sqrt(x)` represents a value, it doesn't make sense to put it on a line by itself in a subroutine call statement such as

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

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

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

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

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

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

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

- `Math.abs(x)`, which computes the absolute value of `x`.
- The usual trigonometric functions, `Math.sin(x)`, `Math.cos(x)`, and `Math.tan(x)`. (For all the trigonometric functions, angles are measured in radians, not degrees.)
- The inverse trigonometric functions `arcsin`, `arccos`, and `arctan`, which are written as: `Math.asin(x)`, `Math.acos(x)`, and `Math.atan(x)`.
- The 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`. (For example, `Math.floor(3.76)` is `3.0`.)
- `Math.random()`, which returns a randomly chosen `double` in the range `0.0 <=`

`Math.random() < 1.0`. (The computer actually calculates so-called "pseudorandom" numbers, which are not truly random but are random enough for most purposes.)

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

Note that `Math.random()` does not have any parameter. You still need the parentheses, even though there's nothing between them. The parentheses let the computer know that this is a subroutine rather than a variable. Another example of a subroutine that has no parameters is the function `System.currentTimeMillis()`, from the `System` class. When this function is executed, it retrieves the current time, expressed as the number of milliseconds that have passed since a standardized base time (the start of the year 1970 in Greenwich Mean Time, if you care). One millisecond is one thousandth second. The value of `System.currentTimeMillis()` is of type `long`. This function can be used to measure the time that it takes the computer to perform a task. Just record the time at which the task is begun and the time at which it is finished and take the difference.

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

```
public class TimedComputation {

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

    public static void main(String[] args) {

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

        startTime = System.currentTimeMillis();

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

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

        System.out.print("\nHere is a random number:  ");
    }
}
```

```

        System.out.println( Math.random() );

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

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

    } // end main()

} // end class TimedComputation

```

Here is a simulated version of this program. If you run it several times, you should see a different random number in the output each time.

(Applet "TimedComputationConsole" would be displayed here
if Java were available.)

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, `length` is a subroutine that computes the length of a string. Suppose that `str` is a variable that refers to a `String`. For example, `str` might have been declared and assigned a value as follows:

```

String str;
str = "Seize the day!";

```

Then `str.length()` is a function call that represents the number of characters in the string. The value of `str.length()` is an `int`. Note that this function has no parameter; the string whose length is being computed is `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.println("the string \"Hello World\" is ");
System.out.println( "Hello World".length() );

```

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

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

starting position of that substring. Otherwise, the returned value is -1. You can also use `s1.indexOf(ch)` to search for a particular character, `ch`, in `s1`. To find the first occurrence of `x` at or after position `N`, you can use `s1.indexOf(x,N)`.

- `s1.compareTo(s2)` is an integer-valued function that compares the two strings. If the strings are equal, the value returned is zero. If `s1` is less than `s2`, the value returned is a number less than zero, and if `s1` is greater than `s2`, the value returned is some number greater than zero. (If both of the strings consist entirely of lowercase letters, then "less than" and "greater than" refer to alphabetical order. Otherwise, the ordering is more complicated.)
- `s1.toUpperCase()` is a `String`-valued function that returns a new string that is equal to `s1`, except that any lower case letters in `s1` have been converted to upper case. For example, `"Cat".toUpperCase()` is the string `"CAT"`. There is also a method `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"`.

For the methods `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 `"s2 = 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.) 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 concatenate values belonging to one of the primitive types onto a `String` using the `+` operator. The value of primitive type is converted to a string, just as it would be if you printed it to the standard output, and then it is concatenated onto the string. For example, the expression `"Number" + 42` evaluates to the string `"Number42"`. And the statements

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

can be replaced by the single statement:

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

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

Section 2.4

Text Input and Output

FOR SOME UNFATHOMABLE REASON, Java seems to lack any reasonable built-in subroutines for reading data typed in by the user. You've already seen that output can be displayed to the user using the subroutine `System.out.print`. This subroutine is part of a pre-defined object called `System.out`. The purpose of this object is precisely to display output to the user. There is a corresponding object called `System.in` that exists to read data input by the user, but it provides only very primitive input facilities, and it requires some advanced Java programming skills to use it effectively.

There is some excuse for this, since Java is meant mainly to write programs for Graphical User Interfaces, and those programs have their own style of input/output, which is implemented in Java. However, basic support is needed for input/output in old-fashioned non-GUI programs. Fortunately, it is possible to **extend Java by creating new classes that provide subroutines that are not available in the classes which are a 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.**

For example, I've written a class called `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 `System.in` directly. `TextIO` also contains a set of output subroutines. The output subroutines are similar to those provided in `System.out`, but they provide a few additional features. You can use whichever set of output subroutines you prefer, and you can even mix them in the same program.

To use the `TextIO` class, you must make sure that the class is available to your program. What this means depends on the Java programming environment that you are using. See [Appendix 2](#) for information about programming environments. In general, you just have to add the compiled file, `TextIO.class`, to the directory that contains your main program. You can obtain the compiled class file by compiling the source code, [TextIO.java](#).

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 `getInt` 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.getInt`. The function has no parameters, so a call to the function takes the form `"TextIO.getInt()"`. 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.getInt();
```

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

```
public class PrintSquare {

    public static void main(String[] args) {

        // A program that computes and prints the square
        // of a number input by the user.
```



```

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

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

    } // end of main()

} //end of class PrintSquare

```

Here's an applet that simulates this program. When you run the program, it will display the message "Please type a number: " and will pause until you type a response. (If the applet does not respond to your typing, you might have to click on it to activate it. In some browsers, you might also need to leave the mouse cursor inside the applet for it to recognize your typing.)

(Applet "PrintSquareConsole" would be displayed here
if Java were available.)

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

```

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

```

with

```

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

```

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

Let's look a little more closely at the built-in output subroutines `System.out.print` and `System.out.println`. Each of these subroutines can be used with one parameter, where the parameter can be any value of type `byte`, `short`, `int`, `long`, `float`, `double`, `char`, `boolean`, or `String`. (These are the eight primitive types plus the type `String`.) That is, you can say "`System.out.print(x);`" or "`System.out.println(x);`", where `x` is any expression whose value is of one of these types. The expression can be a constant, a variable, or even something more complicated such as `2*distance*time`. In fact, there are actually several different subroutines to handle the different parameter types. There is one `System.out.print` for printing values of type `double`, one for values of type `int`, another for values of type `String`, and so on. These subroutines can have the same name since the computer can tell which one you mean in a given subroutine call statement, depending on the type of parameter that you supply. Having several subroutines of the same name that differ in the types of their parameters is called **overloading**. Many programming languages do not permit overloading, but it is common in Java programs.

The difference between `System.out.print` and `System.out.println` is that `System.out.println` outputs a carriage return after it outputs the specified parameter value. There is a version of `System.out.println` that has no parameters. This version simply outputs a carriage return, and nothing else. Of course, a subroutine call statement for this version of the program looks like "`System.out.println();`", with empty parentheses. Note that "`System.out.println(x);`" is

exactly equivalent to `"System.out.print(x); System.out.println();"`. (There is no version of `System.out.print` without parameters. Do you see why?)

As mentioned above, the `TextIO` subroutines `TextIO.put` and `TextIO.putln` can be used as replacements for `System.out.print` and `System.out.println`. However, `TextIO` goes beyond `System.out` by providing additional, two-parameter versions of `put` and `putln`. You can use subroutine call statements of the form `"TextIO.put(x,n);"` and `"TextIO.putln(x,n);"`, where the second parameter, `n`, is an integer-valued expression. The idea is that `n` is the number of characters that you want to output. If `x` takes up fewer than `n` characters, then the computer will add some spaces at the beginning to bring the total up to `n`. (If `x` already takes up more than `n` characters, the computer will just print out more characters than you ask for.) This feature is useful, for example, when you are trying to output neat columns of numbers, and you know just how many characters you need in each column.

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

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

```
b = TextIO.getByte();    // value read is a byte
i = TextIO.getShort();   // value read is a short
j = TextIO.getInt();     // value read is an int
k = TextIO.getLong();    // value read is a long
x = TextIO.getFloat();   // value read is a float
y = TextIO.getDouble();  // value read is a double
a = TextIO.getBoolean(); // value read is a boolean
c = TextIO.getChar();    // value read is a char
w = TextIO.getWord();    // value read is a String
s = TextIO.getln();      // value read is a String
```

For these statements to be legal, the variables on the left side of each assignment statement must be of the same type as that returned by the function on the right side.

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 a `byte` and the user types in a number that is outside the legal range of -128 to 127 -- 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.

You'll notice that there are two input functions that return `Strings`. The first, `getWord()`, returns a string consisting of non-blank characters only. When it is called, it skips over any spaces and carriage returns typed in by the user. Then it reads non-blank characters until it gets to the next space or carriage return. It returns a `String` consisting of all the non-blank characters that it has read. The second input function, `getln()`, simply returns a string consisting of all the characters typed in by the user, including spaces, up to the next carriage return. It gets an entire line of input text. The carriage return itself is not returned as part of the input string, but it is read and discarded by the computer. Note that the `String` returned by this function might be the **empty string**, `" "`, which contains no characters at all.

All the other input functions listed -- `getByte()`, `getShort()`, `getInt()`, `getLong()`, `getFloat()`, `getDouble()`, `getBoolean()`, and `getChar()` -- behave like `getWord()`. That is, they will skip past any blanks and carriage returns in the input before reading a value. However, they will not skip past other characters. If you try to read two `ints` and the user types `"2,3"`, the computer will read the first number correctly, but when it tries to read the second number, it will see the comma. It will regard this as an error and will force the user to retype the number. If you want to input several numbers from one line, you should make sure that the user knows to separate them with spaces, not commas. Alternatively, if

you want to require a comma between the numbers, use `getChar()` to read the comma before reading the second number.

There is another character input function, `TextIO.getAnyChar()`, which does not skip past blanks or carriage returns. It simply reads and returns the next character typed by the user. This could be any character, including a space or a carriage return. If the user typed a carriage return, then the `char` returned by `getChar()` 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 semantics of input is much more complicated than the semantics of output. 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 a chunk of internal memory called the **input buffer**. Input is actually read from the buffer, not directly from the user's typing. The user only gets to type when the buffer is empty. This lets you read several numbers from one line of input. However, if you only want to read in one number and the user types in extra stuff on the line, then you could be in trouble. The extra stuff will still be there the next time you try to read something from input. (The symptom of this trouble is that the computer doesn't pause where you think it should to let the user type something in. The computer had stuff left over in the input buffer from the previous line that the user typed.) To help you avoid this, there are versions of the `TextIO` input functions that read a data value and then discard any leftover stuff on the same line:

```
b = TextIO.getlnByte();      // value read is a byte
i = TextIO.getlnShort();    // value read is a short
j = TextIO.getlnInt();      // value read is an int
k = TextIO.getlnLong();     // value read is a long
x = TextIO.getlnFloat();    // value read is a float
y = TextIO.getlnDouble();   // value read is a double
a = TextIO.getlnBoolean();  // value read is a boolean
c = TextIO.getlnChar();     // value read is a char
w = TextIO.getlnWord();     // value read is a String
```

Note that calling `getlnDouble()`, for example, is equivalent to first calling `getDouble()` and then calling `getln()` to read any remaining data on the same line, including the end-of-line character itself. 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.

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

Using `TextIO` for input and output, we can now improve the program from [Section 2](#) 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 more than once!

```
public class Interest2 {

    /*
       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 static void main(String[] args) {

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

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

        TextIO.put("Enter the annual interest rate: ");
        rate = TextIO.getlnDouble();

        interest = principal * rate;    // compute this year's interest
        principal = principal + interest;    // add it to principal

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

    } // end of main()

} // end of class Interest2

```

Try out an equivalent applet here. (If the applet does not respond to your typing, you might have to click on it to activate it.)

(Applet "Interest2Console" would be displayed here
if Java were available.)

By the way, the applets on this page don't actually use `TextIO`. The `TextIO` class is only for use in programs, not applets. For applets, I have written a separate class that provides similar input/output capabilities in a Graphical User Interface program.

[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

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 the output value in an output routine, or combined with other values into a more complicated expression. (The value can even, in some cases, be ignored, if that's what you want to do; this is more common than you might think.) Expressions are an essential part of programming. So far, these notes have dealt only informally with expressions. This section tells you the more-or-less complete story.

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

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

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

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`. 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`. (The computer's internal representations for `10` and `10.0` are very different, even though people think of them as representing the same number.) Ordinarily, you don't have to worry about type conversion, 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 integer is divided by another. This

operator is indicated by `%`. If `A` and `B` are integers, then `A % B` represents the remainder when `A` is divided by `B`. For example, `7 % 2` is 1, while `34577 % 100` is 77, and `50 % 8` is 2. A common use of `%` is to test whether a given integer is even or odd. `N` is even if `N % 2` is zero, and it is odd if `N % 2` is 1. More generally, you can check whether an integer `N` is evenly divisible by an integer `M` by checking whether `N % M` is zero.

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

Increment and Decrement

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

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

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

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

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

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

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

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

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

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 relation operators in Java are: `==`, `!=`, `<`, `>`, `<=`, and `>=`. The meanings of these operators are:

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

These operators can be used to compare values of any of the numeric types. They can also be used to compare values of type `char`. For characters, `<` and `>` are defined according the numeric Unicode values of the characters. (This might not always be what you want. It is not the same as alphabetical order because all the upper case letters come before all the lower case letters.)

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

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

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

One thing that you cannot do with the relational operators `<`, `>`, `<=`, and `>=` is to use them to compare values of type `String`. You can legally use `==` and `!=` to compare `Strings`, but because of peculiarities in the way objects behave, they might not give the results you want. (The `==` operator checks whether two objects are stored in the same memory location, rather than whether they contain the same value. Occasionally, for some objects, you do want to make such a check -- but rarely for strings. I'll get back to this in a later chapter.) Instead, you should use the subroutines `equals()`, `equalsIgnoreCase()`, and `compareTo()`, which were described in [Section 3](#), to compare two `Strings`.

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 illegal, since division by zero is not allowed. However, the computer will never perform the division, since when the computer evaluates `(x != 0)`, it finds that the result is `false`, and so it knows that `((x != 0) && anything)` has to be false. Therefore, it doesn't bother to evaluate the second operand, `(y/x > 1)`. The evaluation has been short-circuited and the division by zero is avoided. Without the short-circuiting, there would have been a division-by-zero error. (This may seem like a technicality, and it is. But at times, it will make your programming life a little easier. To be even more technical: There are actually non-short-circuited versions of `&&` and `||`, which are written as `&` and `|`. Don't use them unless you have a particular reason to do so.)

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`.

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 ? expression-1 : expression-2
```

The computer tests the value of **boolean-expression**. If the value is `true`, it evaluates **expression-1**; otherwise, it evaluates **expression-2**. 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.

Assignment Operators and Type-Casts

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

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

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

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

```

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

```

The idea is that conversion should only be done automatically when it can be done without changing the semantics of the value. Any `int` can be converted to a `double` with the same numeric value. However, there are `int` values that lie outside the legal range of `short`s. 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 34464. (The 34464 is obtained by taking the 4-byte `int` 100000 and throwing away two of those bytes to obtain a `short` -- you've lost the real information that was in those two bytes.)

As another example of type casts, consider the problem of getting a random integer between 1 and 6. The function `Math.random()` gives a real number between 0.0 and 0.9999..., and so `6*Math.random()` is between 0.0 and 5.999.... The type-cast operator, `(int)`, can be used to convert this to an integer: `(int)(6*Math.random())`. A real number is cast to an integer by discarding the fractional part. Thus, `(int)(6*Math.random())` is one of the integers 0, 1, 2, 3, 4, and 5. To get a number between 1 and 6, we can add 1: `"(int)(6*Math.random()) + 1"`.

You can also type-cast between the type `char` and the numeric types. The numeric value of a `char` is its Unicode code number. For example, `(char)97` is `'a'`, and `(int)'+'` is 43.

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

```

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

```

The combined assignment operator `+=` even works with strings. You will recall from [Section 3](#) that when the `+` operator is used with a string as the first operand, 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"`.

Precedence Rules

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

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

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

Operators on the same line have the same precedence. When they occur together, unary operators and assignment operators are evaluated right-to-left, and 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 ?)

End of Chapter 2

[[Next Chapter](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

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:

[illegible]

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 5](#), The expression

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

(Note: The word "dice" is a plural, as in "two dice." The singular is "die.")

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!".

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!](#)

[[Chapter Index](#) | [Main Index](#)]

Quiz Questions For Chapter 2

THIS PAGE CONTAINS A SAMPLE quiz on material from [Chapter 2](#) of this [on-line Java textbook](#). You should be able to answer these questions after studying that chapter. Sample answers to all the quiz questions can be found [here](#).

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();"`

[[Answers](#) | [Chapter Index](#) | [Main Index](#)]

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, loop and branches, can be used to repeat a sequence of statements over and over or to choose among two or more possible courses of action. Java includes several control structures of each type, and we will look at each of them in some detail.

This chapter will also begin the study of program design. Given a problem, how can you come up with a program to solve that problem? We'll look at a partial answer to this question in Section 2. In the following sections, we'll apply the techniques from Section 2 to a variety of examples.

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 Applets and Graphics](#)
 - [Programming Exercises](#)
 - [Quiz on this Chapter](#)
-

[[First Section](#) | [Next Chapter](#) | [Previous Chapter](#) | [Main Index](#)]

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 -- and, in fact, just three of them would be enough to write programs to perform any task. The six control structures are: the **block**, the **while loop**, the **do..while loop**, the **for loop**, the **if statement**, and the **switch statement**. Each of these structures is considered to be a single "statement," but each is in fact a **structured statement that can contain one or more other statements inside itself**.

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

Here are two examples of blocks:

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

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

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

block, and more specifically to the part of the block that comes after the declaration of the variable.

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

A **while loop** is used to repeat a given statement over and over. Of course, it's not likely that you would want to keep repeating it forever. That would be an **infinite loop**, which is generally a bad thing. (There is an old story about computer pioneer Grace Murray Hopper, who read instructions on a bottle of shampoo telling her to "lather, rinse, repeat." As the story goes, she claims that she tried to follow the directions, but she ran out of shampoo. (In case you don't get it, this is a joke about the way that computers mindlessly follow instructions.))

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

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

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

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

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

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

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

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

printed was 5.

By the way, you should remember that you'll never see a `while` loop standing by itself in a real program. It will always be inside a subroutine which is itself defined inside some class. As an example of a `while` loop used inside a complete program, here is a little program that computes the interest on an investment over several years. This is an improvement over examples from the previous chapter that just reported the results for one year:

```
public class Interest3 {

    /*
     * 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 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. */

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

        TextIO.put("Enter the annual interest rate: ");
        rate = TextIO.getlnDouble();

        /* 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.println(principal);
        } // end of while loop

    } // end of main()

} // end of class Interest3
```

And here is the applet which simulates this program:

(Applet "Interest3Console" would be displayed here)

if Java were available.)

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

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

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

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

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

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

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

Of course, either or both of the **statement**'s in an `if` statement can be a block, so that 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.println(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!

[[Next Section](#) | [Previous Chapter](#) | [Chapter Index](#) | [Main Index](#)]

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 terminates after a finite number of steps; we don't want to count procedures that go on forever.) An algorithm is not the same as a program. A program is written in some particular programming language. An algorithm is more like the **idea behind the program, but it's the idea of the steps the program will take to perform its task, not just the idea of the task itself**. The steps of the algorithm don't have to be filled in in complete detail, as long as the steps are unambiguous and it's clear that carrying out the steps will accomplish the assigned task. An algorithm can be expressed in any language, including English. Of course, an algorithm can only be expressed as a program if all the details have been filled in.

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

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

Suppose you have a task in mind that you want the computer to perform. One way to proceed is to write a description of the task, and take that description as an outline of the algorithm you want to develop. Then you can refine and elaborate that description, gradually adding steps and detail, until you have a complete algorithm that can be translated directly into programming language. This method is called **stepwise refinement**, and it is a type of top-down design. As you proceed through the stages of stepwise refinement, you can write out descriptions of your algorithm in **pseudocode** -- informal instructions that imitate the structure of programming languages without the complete detail and perfect syntax of actual program code.

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

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



```

Compute the value after 4 years
Display the value
Compute the value after 5 years
Display the value

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

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

This still needs to be wrapped inside a complete program, it still needs to be commented, and it really needs to print out more information for the user. But it's essentially the same program as the one in the previous section. (Note that the pseudocode algorithm uses indentation to show which statements are inside the loop. In Java, indentation is completely ignored by the computer, so you need a pair of braces to tell the computer which statements are in the loop. If you leave out the braces, the only statement inside the loop would be `years = years + 1;`. The other statements would only be executed once, after the loop ends. The nasty thing is that the computer won't notice this error for you, like it would if you left out the parentheses around `(years < 5)`. The parentheses are required by the syntax of the `while` statement. The braces are only required semantically. The computer can recognize syntax errors but not semantic errors.)

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

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

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

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

"Write a program that will read a positive integer from the user and will print out the $3N+1$ sequence starting from that integer. The program should also count and print out the number of terms in the sequence."

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

```

        else
            Compute N = 3 * N + 1;
            Output N;
            Add 1 to counter;
        Output the counter;

```

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

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

```

public class ThreeN {

    /* 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 static void main(String[] args) {

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

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

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

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

```

```

    }    // end of main()

}    // end of class ThreeN

```

As usual, you can try this out in an applet that simulates the program. Try different starting values for N , including some negative values:

(Applet "ThreeN1Console" would be displayed here
if Java were available.)

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

```

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

```

Second, there is the question of why this problem is at all interesting. Well, it's interesting to mathematicians and computer scientists because of a simple question about the problem that they haven't been able to answer: Will the process of computing the $3N+1$ sequence finish after a finite number of steps for all possible starting values of N ? Although individual sequences are easy to compute, no one has been able to answer the general question. (To put this another way, no one knows whether the process of computing $3N+1$ sequences can properly be called an algorithm, since an algorithm is required to terminate after a finite number of steps!)

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 will 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 rarely 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 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.

[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Section 3.3

The while and do..while Statements

STATEMENTS IN JAVA CAN BE either simple statements or compound statements. Simple statements, such as assignments 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 four sections explore the details of all the 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](#).

The while Statement

The while statement was already introduced in [Section 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 does not, 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
Let count = 0
while there are more integers to process:
    Read an integer
    Add it to the sum
    Count it
Divide sum by count to get the average
Print out the average
```

But how can we test whether there are more integers to process? A typical solution is to tell the user to type in zero after all the data have been entered. This will work because we are assuming that all the data are positive numbers, so zero is not a legal data value. The zero is not itself part of the data to be averaged. It's just there to mark the end of the real data. A data value used in this way is sometimes called a **sentinel value**. So now the test in the while loop becomes "while the input integer is not zero". But there is another problem! The first time the test is evaluated, before the body of the loop has ever been executed, no integer has yet been read. There is no "input integer" yet, so testing whether the input integer is zero doesn't make

sense. So, we have to do something before the while loop to make sure that the test makes sense. Setting things up so that the test in a while loop makes sense the first time it is executed is called **priming the loop**. In this case, we can simply read the first integer before the beginning of the loop. Here is a revised algorithm:

```

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

```

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

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

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

Here is the program and an applet that simulates it:

```

public class ComputeAverage {

    /* 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 work for
       both positive and negative input values.
    */

    public static void main(String[] args) {

        int inputNumber;    // One of the integers input by the user.
        int sum;            // The sum of the positive integers.
    }
}

```

```

    int count;           // The number of positive integers.
    double average;      // The average of the positive integers.

    /* Initialize the summation and counting variables. */

    sum = 0;
    count = 0;

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

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

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

    /* Display the result. */

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

} // end main()

} // end class ComputeAverage

```

(Applet "ComputeAverageConsole" would be displayed here
if Java were available.)

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 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 executed at least once.

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

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

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

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

When the value of the boolean variable is set to `true`, 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.

The break and continue Statements

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

```
break;
```

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

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

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

(The first line of the 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 can be terminated only 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. I won't give the details here; you can look them up if you ever need them.

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 (after evaluating the loop's continuation condition to see whether any further iterations are required).

`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 6](#), we'll see that `break` can also be used to break out of a `switch` statement. Note that when a `break` occurs inside an `if` statement, it breaks out of the loop or `switch` statement that contains the `if` statement. If the `if` statement is not contained inside a loop or `switch`, then the `if` statement cannot legally contain a `break` statement. A similar consideration applies to `continue` statements.

[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

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.

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 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 makes the loop easier to read and understand. The `for` loop is executed in exactly the same way as the original code: The initialization part is executed once, before the loop begins. The continuation condition is executed before each execution of the loop, and the loop ends when this condition is `false`. The update part is executed at the end of each execution of the loop, just before jumping back to check the condition.

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

```

for ( initialization; continuation-condition; update )
    statement

```

or, using a block statement:

```

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

```

The **continuation-condition** must be a boolean-valued expression. The **initialization** can be any expression,

as can the **update**. Any of the three can be empty. If the continuation condition is empty, it is treated as if it were "true," so the loop will be repeated forever or until it ends for some other reason, such as a `break` statement. (Some people like to begin an infinite loop with "`for (; ;)`" instead of "`while (true)`".)

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

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

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

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

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

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

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

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

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

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

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

```
for ( i=1, j=10; i <= 10; i++, j-- ) {
    TextIO.put(i,5);    // Output i in a 5-character wide column.
    TextIO.putln(j,5);  // Output j in a 5-character column
                        // and end the line.
}
```

As a final 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.print("2 4 6 8 10 12 ");
            System.out.println("14 16 18 20");
        }

```

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

```

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

```

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

N is an even multiple of D .) In terms of Java programming, D is a divisor of N if $N \% D$ is zero.

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

Finally, we can translate the algorithm into a complete Java program. Here it is, followed by an applet that simulates it:

```

public class CountDivisors {

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

```

```

    */

    public static void main(String[] args) {

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

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

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

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

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

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

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

        divisorCount = 0;
        numberTested = 0;

        for (testDivisor = 1; testDivisor <= N; testDivisor++) {
            if ( N % testDivisor == 0 )
                divisorCount++;
            numberTested++;
            if (numberTested == 1000000) {
                TextIO.put('.');
                numberTested = 0;
            }
        }

        /* Display the result. */

        TextIO.putln();
        TextIO.putln("The number of divisors of " + N
                    + " is " + divisorCount);

    } // end main()
} // end class CountDivisors

```

**(Applet "CountDivisorsConsole" would be displayed here
if Java were available.)**

Nested Loops

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

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

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

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

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

The first step in the `for` loop can itself be expressed as a `for` loop:

```
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
```

Assuming that `rowNumber` and `N` have been declared to be variables of type `int`, this can be expressed in Java as

```
for ( rowNumber = 1; rowNumber <= 12; rowNumber++ ) {
    for ( N = 1; N <= 12; N++ ) {
        // print in 4-character columns
        TextIO.put( N * rowNumber, 4 );
    }
    TextIO.putln();
}
```

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

number of different letters. The string will be input by the user. Let's start with a pseudocode algorithm for the program.

```

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

```

Since we want to process the entire line of text that is entered by the user, we'll use `TextIO.getln()` to read it. The line of the algorithm that reads "for each letter of the alphabet" can be expressed as "for (letter='A'; letter<='Z'; letter++)". But the body of this for loop needs more thought. How do we check whether the given letter, `letter`, occurs in `str`? One idea is to look at each letter in the string in turn, and check whether that letter 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. Note the use of `break` in the nested for loop. It is required to avoid printing or counting a given letter more than once. 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`.

```

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
Output the count

```

Here is the complete program and an applet to simulate it:

```

public class ListLetters {

    /* 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 static void main(String[] args) {

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

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

```

```

        str = str.toUpperCase();

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

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

    } // end main()
} // end class ListLetters

```

(Applet "ListLettersConsole" would be displayed here
if Java were available.)

In fact, there is 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 preview of how subroutines can be used to deal with complexity.

[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

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 1](#). It takes the form

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

As usual, the statements inside an `if` statements 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.

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 statements have different meanings: If $x \leq 0$, the first statement doesn't print anything, but the second statement prints "Second case."

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 multi-way branches with any number of cases:

```

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

```

The computer evaluates boolean expressions one after the other until it comes to one that is `true`. It executes the associated statement and skips the rest. If none of the boolean expressions evaluate to `true`, then the statement in the `else` part is executed. This statement is called a multi-way branch because only one of the statements will be executed. The final `else` part can be omitted. In that case, if all the boolean expressions are `false`, none of the statements is 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.)

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 examples, if the values are 42, 17, and 20, then the output should be in the order 17, 20, 42.

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

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

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

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

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

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

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

```
if ( x < y ) { // x comes before y
    if ( z < x )
        System.out.println( z + " " + x + " " + y );
}
```

```

        else if ( z > y )
            System.out.println( x + " " + y + " " + z );
        else
            System.out.println( x + " " + z + " " + y );
    }
    else {
        // y comes before x
        if ( z < y )
            System.out.println( z + " " + y + " " + x );
        else if ( z > x )
            System.out.println( y + " " + x + " " + z );
        else
            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 units of measure. The conversion into different units of measure can be simplified by first converting the user's input into inches. From there, it can be converted into feet, yards, and miles. We still 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 enter a whole sequence of measurements. The program will end only when the user inputs 0. To do this, I just have to wrap the above algorithm inside a `while` loop, and make sure that the loop ends when the user inputs a 0. Here's the complete program, followed by an applet that simulates it.

```
public class LengthConverter {

    /* 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",
       "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 static void main(String[] args) {

        double measurement; // Numerical measurement, input by user.
        String units;        // The unit of measure for the input, also
                            // specified by the user.

        double inches, feet, yards, miles; // Measurement expressed in
                                           // each possible unit of
                                           // measure.

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

        while (true) {

            /* Get the user's input, and convert units to lower case. */

            TextIO.put("Enter your measurement, or 0 to end:  ");
            measurement = TextIO.getDouble();
            if (measurement == 0)
                break; // terminate the while loop
            units = TextIO.getlnWord();
            units = units.toLowerCase();

            /* Convert the input measurement to inches. */

            if (units.equals("inch") || units.equals("inches")
                || units.equals("in")) {
                inches = measurement;
            }
        }
    }
}
```

```

        else if (units.equals("foot") || units.equals("feet")
                || units.equals("ft")) {
            inches = measurement * 12;
        }
        else if (units.equals("yard") || units.equals("yards")
                || units.equals("yd")) {
            inches = measurement * 36;
        }
        else if (units.equals("mile") || units.equals("miles")
                || units.equals("mi")) {
            inches = measurement * 12 * 5280;
        }
        else {
            TextIO.putln("Sorry, but I don't understand \""
                    + units + "\".");
            continue; // back to start of while loop
        }

        /* Convert measurement in inches to feet, yards, and miles. */

        feet = inches / 12;
        yards = inches / 36;
        miles = inches / (12*5280);

        /* Output measurement in terms of each unit of measure. */

        TextIO.putln();
        TextIO.putln("That's equivalent to:");
        TextIO.put(inches, 15);
        TextIO.putln(" inches");
        TextIO.put(feet, 15);
        TextIO.putln(" feet");
        TextIO.put(yards, 15);
        TextIO.putln(" yards");
        TextIO.put(miles, 15);
        TextIO.putln(" miles");
        TextIO.putln();

    } // end while

    TextIO.putln();
    TextIO.putln("OK! Bye for now.");

} // end main()

} // end class LengthConverter

```

(Applet "LengthConverterConsole" would be displayed here
if Java were available.)

[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Section 3.6

The switch Statement

THE SECOND BRANCHING STATEMENT in Java is the `switch` statement, which is introduced in this section. The `switch` is used far less often than the `if` statement, but it is sometimes useful for expressing a certain type of multi-way branch. Since this section wraps up coverage of all of Java's control statements, I've included a complete list of Java's statement types at the end of the section.

A `switch` statement allows you to test the value of an expression and, depending on that value, to jump to some location within the `switch` statement. The expression must be either integer-valued or character-valued. It **cannot be a `String` or a real number**. The positions that you can jump to are marked with "case labels" that take the form: "case **constant**:". This marks the position the computer jumps to when the expression evaluates to the given **constant**. As the final case in a `switch` statement you can, optionally, use the label "default:", which provides a default jump point that is used when the value of the expression is not listed in any case label.

A `switch` statement has the form:

```
switch (expression) {
    case constant-1:
        statements-1
        break;
    case constant-2:
        statements-2
        break;
    .
    .    // (more cases)
    .
    case constant-N:
        statements-N
        break;
    default: // optional default case
        statements-(N+1)
} // end of switch statement
```

The `break` statements are technically optional. The effect of a `break` is to make the computer jump to the end of the `switch` statement. If you leave out the `break` statement, the computer will just forge ahead after completing one case and will execute the statements associated with the next case label. This is rarely what you want, but it is legal. (I will note here -- although you won't understand it until you get to the next chapter -- that inside a subroutine, the `break` statement is sometimes replaced by a `return` statement.)

Note that you can leave out one of the groups of statements entirely (including the `break`). You then have two case labels in a row, containing two different constants. This just means that the computer will jump to the same place and perform the same action for each of the two constants.

Here is an example of a `switch` statement. This is not a useful example, but it should be easy for you to follow. Note, by the way, that the constants in the case labels don't have to be in any particular order, as long as they are all different:

```
switch (N) {    // assume N is an integer variable
    case 1:
        System.out.println("The number is 1.");
        break;
    case 2:
```



```

        case 4:
        case 8:
            System.out.println("The number is 2, 4, or 8.");
            System.out.println("(That's a power of 2!)");
            break;
        case 3:
        case 6:
        case 9:
            System.out.println("The number is 3, 6, or 9.");
            System.out.println("(That's a multiple of 3!)");
            break;
        case 5:
            System.out.println("The number is 5.");
            break;
        default:
            System.out.println("The number is 7,");
            System.out.println("    or is outside the range 1 to 9.");
    }

```

The switch statement is pretty primitive as control structures go, and it's easy to make mistakes when you use it. Java takes all its control structures directly from the older programming languages C and C++. The switch statement is certainly one place where the designers of Java should have introduced some improvements.

One application of switch statements is in processing menus. A menu is a list of options. The user selects one of the options. The computer has to respond to each possible choice in a different way. If the options are numbered 1, 2, ..., then the number of the chosen option can be used in a switch statement to select the proper response.

In a TextIO-based program, the menu can be presented as a numbered list of options, and the user can choose an option by typing in its number. Here is an example that could be used in a variation of the LengthConverter example from the [previous section](#):

```

int optionNumber;    // Option number from menu, selected by user.
double measurement; // A numerical measurement, input by the user.
                    // The unit of measurement depends on which
                    // option the user has selected.
double inches;       // The same measurement, converted into inches.

/* Display menu and get user's selected option number. */

TextIO.putln("What unit of measurement does your input use?");
TextIO.putln();
TextIO.putln("    1.  inches");
TextIO.putln("    2.  feet");
TextIO.putln("    3.  yards");
TextIO.putln("    4.  miles");
TextIO.putln();
TextIO.putln("Enter the number of your choice: ");
optionNumber = TextIO.getlnInt();

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

switch ( optionNumber ) {
    case 1:

```

```

        TextIO.putln("Enter the number of inches: ");
        measurement = TextIO.getlnDouble();
        inches = measurement;
        break;
    case 2:
        TextIO.putln("Enter the number of feet: ");
        measurement = TextIO.getlnDouble();
        inches = measurement * 12;
        break;
    case 3:
        TextIO.putln("Enter the number of yards: ");
        measurement = TextIO.getlnDouble();
        inches = measurement * 36;
        break;
    case 4:
        TextIO.putln("Enter the number of miles: ");
        measurement = TextIO.getlnDouble();
        inches = measurement * 12 * 5280;
        break;
    default:
        TextIO.putln("Error!  Illegal option number!  I quit!");
        System.exit(1);
} // end switch

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

```

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. 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". I prefer, though, to use an empty block, consisting of `{` and `}` with nothing between, for such cases.

Occasionally, stray empty statements can cause annoying, hard-to-find errors in a program. For example, the following program segment prints out "Hello" just once, not ten times:

```

for (int i = 0; i < 10; i++);
    System.out.println("Hello");

```

Why? Because the `;` at the end of the first line is a statement, and it is this statement that is executed ten times. The `System.out.println` statement is not really inside the `for` statement at all, so it is executed just once, after the `for` loop has completed.

A List of Java Statement Types

I mention the empty statement here mainly for completeness. You've now seen just about every type of Java statement. A complete list is given below for reference. The only new items in the list are the `try..catch`, `throw`, and `synchronized` statements, which are related to advanced aspects of Java known as exception-handling and multi-threading, and the `return` statement, which is used in subroutines. These will be covered in later sections.

Another possible surprise is what I've listed as "other expression statement," which reflects the fact that any expression followed by a semicolon can be used as a statement. To execute such a statement, the computer simply evaluates the expression, and then ignores the value. Of course, this only makes sense when the evaluation has a **side effect** that makes some change in the state of the computer. An example of this is the expression statement `"x++;"`, which has the side effect of adding 1 to the value of `x`. Similarly, the function call `"TextIO.getln()"`, which reads a line of input, can be used as a stand-alone statement if you want to read a line of input and discard it. Note that, technically, assignment statements and subroutine call statements are also considered to be expression statements.

Java statement types:

- declaration statement (for declaring variables)
- assignment statement
- subroutine call statement (including input/output routines)
- other expression statement (such as `"x++;"`)
- empty statement
- block statement
- while statement
- do..while statement
- if statement
- for statement
- switch statement
- break statement (found in loops and switch statements only)
- continue statement (found in loops only)
- return statement (found in subroutine definitions only)
- try..catch statement
- throw statement
- synchronized statement

[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Section 3.7

Introduction to Applets and Graphics

FOR THE PAST TWO CHAPTERS, you've been learning the sort of programming that is done inside a single subroutine. In the rest of the text, we'll be more concerned with the larger scale structure of programs, but the material that you've already learned will be an important foundation for everything to come.

In this section, before moving on to programming-in-the-large, we'll take a look at how programming-in-the-small can be used in other contexts besides text-based, command-line-style programs. We'll do this by taking a short, introductory look at applets and graphical programming.

An **applet** is a Java program that runs on a Web page. An applet is not a stand-alone application, and it does not have a `main()` routine. In fact, an applet is an **object rather than a class**. When an applet is placed on a Web page, it is assigned a rectangular area on the page. It is the job of the applet to draw the contents of that rectangle. When the region needs to be drawn, the Web page calls a subroutine in the applet to do so. This is not so different from what happens with stand-alone programs. When a program needs to be run, the system calls the `main()` routine of the program. Similarly, when an applet needs to be drawn, the Web page calls the `paint()` routine of the applet. The programmer specifies what happens when these routines are called by filling in the bodies of the routines. Programming in the small! Applets can do other things besides draw themselves, such as responding when the user clicks the mouse on the applet. Each of the applet's behaviors is defined by a subroutine in the applet object. The programmer specifies how the applet behaves by filling in the bodies of the appropriate subroutines.

A very simple applet, which does nothing but draw itself, can be defined by a class that contains nothing but a `paint()` routine. The source code for the class would have the form:

```
import java.awt.*;
import java.applet.*;

public class name-of-applet extends Applet {

    public void paint(Graphics g) {
        statements
    }

}
```

where **name-of-applet** is an identifier that names the class, and the **statements** are the code that actually draws the applet. This looks similar to the definition of a stand-alone program, but there are a few things here that need to be explained, starting with the first two lines.

When you write a program, there are certain built-in classes that are available for you to use. These built-in classes include `System` and `Math`. If you want to use one of these classes, you don't have to do anything special. You just go ahead and use it. But Java also has a large number of standard classes that are there if you want them but that are not automatically available to your program. (There are just too many of them.) If you want to use these classes in your program, you have to ask for them first. The standard classes are grouped into so-called "packages." Two of these packages are called "java.awt" and "java.applet". The directive "import java.awt.*;" makes all the classes from the package java.awt available for use in your program. The java.awt package contains classes related to graphical user interface programming, including a class called `Graphics`. The `Graphics` class is referred to in the `paint()` routine above. The java.applet package contains classes specifically related to applets, including the class named `Applet`.

The first line of the class definition above says that the class "extends `Applet`." `Applet` is a standard

class that is defined in the `java.applet` package. It defines all the basic properties and behaviors of applet objects. By extending the `Applet` class, the new class we are defining inherits all those properties and behaviors. We only have to define the ways in which our class differs from the basic `Applet` class. In our case, the only difference is that our applet will draw itself differently, so we only have to define the `paint()` routine. This is one of the main advantages of object-oriented programming.

(Actually, most of our applets will be defined to extend `JApplet` rather than `Applet`. The `JApplet` class is itself an extension of `Applet`. The `Applet` class has existed since the original version of Java, while `JApplet` is part of the newer "Swing" set of graphical user interface components. For the moment, the distinction is not important.)

One more thing needs to be mentioned -- and this is a point where Java's syntax gets unfortunately confusing. Applets are objects, not classes. Instead of being static members of a class, the subroutines that define the applet's behavior are part of the applet object. We say that they are "non-static" subroutines. Of course, objects are related to classes because every object is described by a class. Now here is the part that can get confusing: Even though a non-static subroutine is not actually part of a class (in the sense of being part of the behavior of the class), it is nevertheless defined in a class (in the sense that the Java code that defines the subroutine is part of the Java code that defines the class). Many objects can be described by the same class. Each object has its own non-static subroutine. But the common definition of those subroutines -- the actual Java source code -- is physically part of the class that describes all the objects. To put it briefly: static subroutines in a class definition say what the class does; non-static subroutines say what all the objects described by the class do. An applet's `paint()` routine is an example non-static subroutine. A stand-alone program's `main()` routine is an example of a static subroutine. The distinction doesn't really matter too much at this point: When working with stand-alone programs, mark everything with the reserved word, "static"; leave it out when working with applets. However, the distinction between static and non-static will become more important later in the course.

Let's write an applet that draws something. In order to write an applet that draws something, you need to know what subroutines are available for drawing, just as in writing text-oriented programs you need to know what subroutines are available for reading and writing text. In Java, the built-in drawing subroutines are found in objects of the class `Graphics`, one of the classes in the `java.awt` package. In an applet's `paint()` routine, you can use the `Graphics` object `g` for drawing. (This object is provided as a parameter to the `paint()` routine when that routine is called.) `Graphics` objects contain many subroutines. I'll mention just three of them here. You'll find more listed in [Section 6.3](#).

`g.setColor(c)`, is called to set the color that is used for drawing. The parameter, `c` is an object belonging to a class named `Color`, another one of the classes in the `java.awt` package. About a dozen standard colors are available as static member variables in the `Color` class. These standard colors include `Color.black`, `Color.white`, `Color.red`, `Color.green`, and `Color.blue`. For example, if you want to draw in red, you would say "`g.setColor(Color.red);`". The specified color is used for all drawing operations up until the next time `setColor` is called.

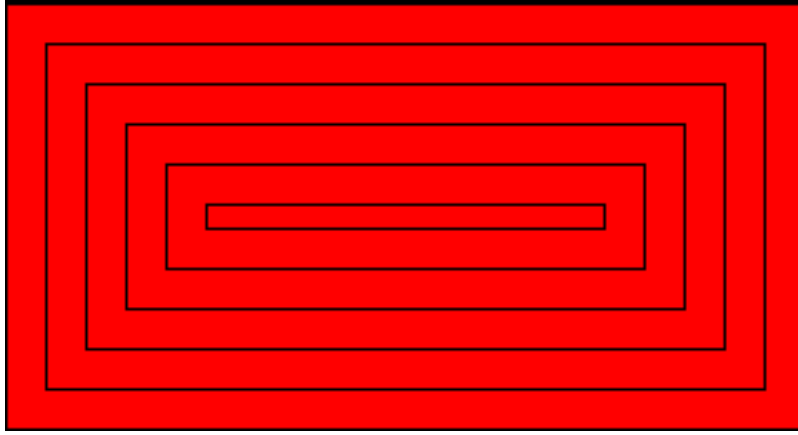
`g.drawRect(x,y,w,h)` draws the outline of a rectangle. The parameters `x`, `y`, `w`, and `h` must be integers. This draws the outline of the rectangle whose top-left corner is `x` pixels from the left edge of the applet and `y` pixels down from the top of the applet. The width of the rectangle is `w` pixels, and the height is `h` pixels.

`g.fillRect(x,y,w,h)` is similar to `drawRect` except that it fills in the inside of the rectangle instead of just drawing an outline.

This is enough information to write the applet shown here:

Sorry, Java is not available.

But here's the picture that the applet draws:



This applet first fills its entire rectangular area with red. Then it changes the drawing color to black and draws a sequence of rectangles, where each rectangle is nested inside the previous one. The rectangles can be drawn with a `while` loop. Each time through the loop, the rectangle gets smaller and it moves down and over a bit. We'll need variables to hold the width and height of the rectangle and a variable to record how far the top-left corner of the rectangle is inset from the edges of the applet. The while loop ends when the rectangle shrinks to nothing. In general outline, the algorithm for drawing the applet is

```
Set the drawing color to red (using the g.setColor subroutine)
Fill in the entire applet (using the g.fillRect subroutine)
Set the drawing color to black
Set the top-left corner inset to be 0
Set the rectangle width and height to be as big as the applet
while the width and height are greater than zero:
    draw a rectangle (using the g.drawRect subroutine)
    increase the inset
    decrease the width and the height
```

In my applet, each rectangle is 15 pixels away from the rectangle that surrounds it, so the `inset` is increased by 15 each time through the `while` loop. The rectangle shrinks by 15 pixels on the left and by 15 pixels on the right, so the width of the rectangle shrinks by 30 each time through the loop. The height also shrinks by 30 pixels each time through the loop.

It is not hard to code this algorithm into Java and use it to define the `paint()` method of an applet. I've assumed that the applet has a height of 160 pixels and a width of 300 pixels. The size is actually set in the source code of the Web page where the applet appears. In order for an applet to appear on a page, the source code for the page must include a command that specifies which applet to run and how big it should be. (The commands that can be used on a Web page are discussed in [Section 6.2](#).) It's not a great idea to assume that we know how big the applet is going to be. On the other hand, it's also not a great idea to write an applet that does nothing but draw a static picture. I'll address both these issues before the end of this section. But for now, here is the source code for the applet:

```
import java.awt.*;
import java.applet.Applet;

public class StaticRects extends Applet {

    public void paint(Graphics g) {

        // Draw a set of nested black rectangles on a red background.
        // Each nested rectangle is separated by 15 pixels on
        // all sides from the rectangle that encloses it.
```

```

    int inset;        // Gap between borders of applet
                      //          and one of the rectangles.

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

    g.setColor(Color.red);
    g.fillRect(0,0,300,160);    // Fill the entire applet with red.

    g.setColor(Color.black);    // Draw the rectangles in black.

    inset = 0;

    rectWidth = 299;    // Set size of first rect to size of applet.
    rectHeight = 159;

    while (rectWidth >= 0 && rectHeight >= 0) {
        g.drawRect(inset, inset, rectWidth, rectHeight);
        inset += 15;        // Rects are 15 pixels apart.
        rectWidth -= 30;    // Width decreases by 15 pixels
                           //          on left and 15 on right.
        rectHeight -= 30;   // Height decreases by 15 pixels
                           //          on top and 15 on bottom.
    }

}    // end paint()

}    // end class StaticRects

```

(You might wonder why the initial `rectWidth` is set to 299, instead of to 300, since the width of the applet is 300 pixels. It's because rectangles are drawn as if with a pen whose nib hangs below and to the right of the point where the pen is placed. If you run the pen exactly along the right edge of the applet, the line it draws is actually outside the applet and therefore is not seen. So instead, we run the pen along a line one pixel to the left of the edge of the applet. The same reasoning applies to `rectHeight`. Careful graphics programming demands attention to details like these.)

When you write an applet, you get to build on the work of the people who wrote the `Applet` class. The `Applet` class provides a framework on which you can hang your own work. Any programmer can create additional frameworks that can be used by other programmers as a basis for writing specific types of applets or stand-alone programs. One example is the applets in previous sections that simulate text-based programs. All these applets are based on a class called `ConsoleApplet`, which itself is based on the standard `Applet` class. You can write your own console applet by filling in this simple framework (which leaves out just a couple of bells and whistles):

```

public class name-of-applet extends ConsoleApplet {

    public void program() {
        statements
    }

}

```

The statements in the `program()` subroutine are executed when the user of the applet clicks the applet's "Run Program" button. This "program" can't use `TextIO` or `System.out` to do input and output. However, the `ConsoleApplet` framework provides an object named `console` for doing text

input/output. This object contains exactly the same set of subroutines as the `TextIO` class. For example, where you would say `TextIO.putln("Hello World")` in a stand-alone program, you could say `console.putln("Hello World")` in a console applet. The `console` object just displays the output on the applet instead of on standard output. Similarly, you can substitute `x = console.getInt()` for `x = TextIO.getInt()`, and so on. As a simple example, here's a console applet that gets two numbers from the user and prints their product:

```
public class PrintProduct extends ConsoleApplet {

    public void program() {

        double x,y;    // Numbers input by the user.
        double prod;   // The product, x*y.

        console.put("What is your first number? ");
        x = console.getlnDouble();
        console.put("What is your second number? ");
        y = console.getlnDouble();

        prod = x * y;
        console.putln();
        console.put("The product is ");
        console.putln(prod);

    } // end program()

} // end class PrintProduct
```

And here's what this applet looks like on a Web page:

(Applet "PrintProduct" would be displayed here
if Java were available.)

Now, any console-style applet that you write depends on the `ConsoleApplet` class, which is not a standard part of Java. This means that the compiled class file, `ConsoleApplet.class` must be available to your applet when it is run. As a matter of fact, `ConsoleApplet` uses two other non-standard classes, `ConsolePanel` and `ConsoleCanvas`, so the compiled class files `ConsolePanel.class` and `ConsoleCanvas.class` must also be available to your applet. This just means that all four class files -- your own class and the three classes it depends on -- must be in the same directory with the source code for the Web page on which your applet appears.

I've written another framework that makes it possible to write applets that display simple animations. An example is given by the applet at the bottom of this page, which is an animated version of the nested squares applet from earlier in this section.

A **computer animation** is really just a sequence of still images. The computer displays the images one after the other. Each image differs a bit from the preceding image in the sequence. If the differences are not too big and if the sequence is displayed quickly enough, the eye is tricked into perceiving continuous motion.

In the example, rectangles shrink continually towards the center of the applet, while new rectangles appear at the edge. The perpetual motion is, of course, an illusion. If you think about it, you'll see that the applet loops through the same set of images over and over. In each image, there is a gap between the borders of the applet and the outermost rectangle. This gap gets wider and wider until a new rectangle appears at the border. Only it's not a new rectangle. What has really happened is that the applet has started over again with the first image in the sequence.

The problem of creating an animation is really just the problem of drawing each of the still images that make up the animation. Each still image is called a **frame**. In my framework for animation, which is based on a non-standard class called `SimpleAnimationApplet2`, all you have to do is fill in the code that says how to draw one frame. The basic format is as follows:

```
import java.awt.*;

public class name-of-class extends SimpleAnimationApplet2 {

    public void drawFrame(Graphics g) {
        statements // to draw one frame of the animation
    }

}
```

The `"import java.awt.*;"` is required to get access to graphics-related classes such as `Graphics` and `Color`. You get to fill in any name you want for the class, and you get to fill in the statements inside the subroutine. The `drawFrame()` subroutine will be called by the system each time a frame needs to be drawn. All you have to do is say what happens when this subroutine is called. Of course, you have to draw a different picture for each frame, and to do that you need to know which frame you are drawing. The class `SimpleAnimationApplet2` provides a function named `getFrameNumber()` that you can call to find out which frame to draw. This function returns an integer value that represents the frame number. If the value returned is 0, you are supposed to draw the first frame; if the value is 1, you are supposed to draw the second frame, and so on.

In the sample applet, the thing that differs from one frame to another is the distance between the edges of the applet and the outermost rectangle. Since the rectangles are 15 pixels apart, this distance increases from 0 to 14 and then jumps back to 0 when a "new" rectangle appears. The appropriate value can be computed very simply from the frame number, with the statement `"inset = getFrameNumber() % 15;"`. The value of the expression `getFrameNumber() % 15` is between 0 and 14. When the frame number reaches 15, the value of `getFrameNumber() % 15` jumps back to 0.

Drawing one frame in the sample animated applet is very similar to drawing the single image of the `StaticRects` applet, as given above. The `paint()` method in the `StaticRects` applet becomes, with only minor modification, the `drawFrame()` method of my `MovingRects` animation applet. I've chosen to make one improvement: The `StaticRects` applet assumes that the applet is 300 by 160 pixels. The `MovingRects` applet will work for any applet size. To implement this, the `drawFrame()` routine has to know how big the applet is. There are two functions that can be called to get this information. The function `getWidth()` returns an integer value representing the width of the applet, and the function `getHeight()` returns the height. The width and height, together with the frame number, are used to compute the size of the first rectangle that is drawn. Here is the complete source code:

```
import java.awt.*;

public class MovingRects extends SimpleAnimationApplet2 {

    public void drawFrame(Graphics g) {

        // Draw one frame in the animation by filling in the background
        // with a solid red and then drawing a set of nested black
        // rectangles. The frame number tells how much the first
        // rectangle is to be inset from the borders of the applet.

        int width;    // Width of the applet, in pixels.
        int height;   // Height of the applet, in pixels.
```

```

        int inset;           // Gap between borders of applet and a rectangle.
                               // The inset for the outermost rectangle goes
                               // from 0 to 14 then back to 0, and so on,
                               // as the frameNumber varies.

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

        width = getWidth();      // Find out the size of the drawing area.
        height = getHeight();

        g.setColor(Color.red);    // Fill the frame with red.
        g.fillRect(0,0,width,height);

        g.setColor(Color.black);  // Switch color to black.

        inset = getFrameNumber() % 15; // Get the inset for the
                                       // outermost rect.

        rectWidth = width - 2*inset - 1; // Set size of outermost rect.
        rectHeight = height - 2*inset - 1;

        while (rectWidth >= 0 && rectHeight >= 0) {
            g.drawRect(inset,inset,rectWidth,rectHeight);
            inset += 15; // Rects are 15 pixels apart.
            rectWidth -= 30; // Width decreases by 15 pixels
                           // on left and 15 on right.
            rectHeight -= 30; // Height decreases by 15 pixels
                           // on top and 15 on bottom.
        }
    } // end drawFrame()

} // end class MovingRects

```

(The [SimpleAnimationApplet2](#) class uses Swing and requires Java version 1.3 or better. There is an older version named [SimpleAnimationApplet](#) that provides the same functionality but works with any version of Java. You could use SimpleAnimationApplet to write animations that will work on older Web browsers.)

The main point here is that by building on an existing framework, you can do interesting things using the type of local, inside-a-subroutine programming that was covered in Chapters 2 and 3. As you learn more about programming and more about Java, you'll be able to do more on your own -- but no matter how much you learn, you'll always be dependent on other people's work to some extent.

End of Chapter 3

[[Next Chapter](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Programming Exercises For Chapter 3

THIS PAGE CONTAINS programming exercises based on material from [Chapter 3](#) of this [on-line Java textbook](#). Each exercise has a link to a discussion of one possible solution of that exercise.

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. One of the examples from [Section 3.4](#) 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 consist of a number, followed by an operator, followed by another number. The operators that are allowed are $+$, $-$, $*$, and $/$. You can read the numbers with `TextIO.getDouble()` and the operator with `TextIO.getChar()`. Your program should read an expression, print its value, read another expression, print its value, and so on. The program should end when the user enters 0 as the first number on the line.

[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
```

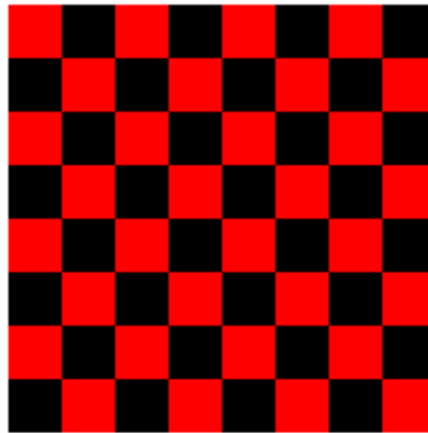
good
idea

(An improved version of the program would list "that's" as a word. An apostrophe can be considered to be part of a word if there is a letter on each side of the apostrophe. But that's not part of the assignment.)

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. For example, it counts an accented e, é, as a letter.

[See the solution!](#)

Exercise 3.5: Write an applet that draws a checkerboard. Assume that the size of the applet is 160 by 160 pixels. Each square in the checkerboard is 20 by 20 pixels. The checkerboard contains 8 rows of squares and 8 columns. The squares are red and black. Here is a tricky way to determine whether a given square is red or black: If the row number and the column number are either both even or both odd, then the square is red. Otherwise, it is black. Note that a square is just a rectangle in which the height is equal to the width, so you can use the subroutine `g.fillRect()` to draw the squares. Here is an image of the checkerboard:



(To run an applet, you need a Web page to display it. A very simple page will do. Assume that your applet class is called `Checkerboard`, so that when you compile it you get a class file named `Checkerboard.class`. Make a file that contains only the lines:

```
<applet code="Checkerboard.class" width=160 height=160>
</applet>
```

Call this file `Checkerboard.html`. This is the source code for a simple Web page that shows nothing but your applet. You can open the file in a Web browser or with Sun's appletviewer program. The compiled class file, `Checkerboard.class`, must be in the same directory with the Web-page file, `Checkerboard.html`.)

[See the solution!](#)

Exercise 3.6: Write an animation applet that shows a checkerboard pattern in which the even numbered rows slide to the left while the odd numbered rows slide to the right. You can assume that the applet is 160 by 160 pixels. Each row should be offset from its usual position by the amount `getFrameNumber() % 40`. Hints: Anything you draw outside the boundaries of the applet will be invisible, so you can draw more than 8 squares in a row. You can use negative values of `x` in `g.fillRect(x,y,w,h)`. Here is a working solution to this exercise:

Your applet will extend the non-standard class, `SimpleAnimationApplet2`, which was introduced in [Section 7](#). When you run your applet, the compiled class files, `SimpleAnimationApplet2.class` and `SimpleAnimationApplet2$1.class`, must be in the same directory as your Web-page source file and the compiled class file for your own class. These files are produced when you compile [SimpleAnimationApplet2.java](#). Assuming that the name of your class is `SlidingCheckerboard`, then the source file for the Web page should contain the lines:

```
<applet code="SlidingCheckerboard.class" width=160 height=160>
</applet>
```

[See the solution!](#)

[[Chapter Index](#) | [Main Index](#)]

Quiz Questions For Chapter 3

THIS PAGE CONTAINS A SAMPLE quiz on material from [Chapter 3](#) of this [on-line Java textbook](#). You should be able to answer these questions after studying that chapter. Sample answers to all the quiz questions can be found [here](#).

Question 1: Explain briefly what is meant by "pseudocode" and how is it useful in the development of algorithms.

Question 2: What is a *block statement*? How are block statements used in Java programs.

Question 3: What is the main difference between a `while` loop and a `do...while` loop?

Question 4: What does it mean to *prime* a loop?

Question 5: Explain what is meant by an *animation* and how a computer displays an animation.

Question 6: 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 7: 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 8: 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 9: 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 10: 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;  
}
```

[[Answers](#) | [Chapter Index](#) | [Main Index](#)]

Chapter 4

Programming in the Large I Subroutines

ONE WAY TO BREAK UP A COMPLEX PROGRAM into manageable pieces is to use **subroutines**. A subroutine consists of the instructions for carrying out a certain task, grouped together and given a name. Elsewhere in the program, that name can be used as a stand-in for the whole set of instructions. As a computer executes a program, whenever it encounters a subroutine name, it executes all the instructions necessary to carry out the task associated with that subroutine.

Subroutines can be used over and over, at different places in the program. A subroutine can even be used inside another subroutine. This allows you to write simple subroutines and then use them to help write more complex subroutines, which can then be used in turn in other subroutines. In this way, very complex programs can be built up step-by-step, where each step in the construction is reasonably simple.

As mentioned in [Section 3.7](#), subroutines in Java can be either static or non-static. This chapter covers static subroutines only. Non-static subroutines, which are used in true object-oriented programming, will be covered in the next chapter.

Contents of Chapter 4:

- Section 1: [Black Boxes](#)
 - Section 2: [Static Subroutines and Static Variables](#)
 - Section 3: [Parameters](#)
 - Section 4: [Return Values](#)
 - Section 5: [Toolboxes, API's, and Packages](#)
 - Section 6: [More on Program Design](#)
 - Section 7: [The Truth about Declarations](#)
 - [Programming Exercises](#)
 - [Quiz on this Chapter](#)
-

[[First Section](#) | [Next Chapter](#) | [Previous Chapter](#) | [Main Index](#)]

Section 4.1

Black Boxes

A SUBROUTINE CONSISTS OF INSTRUCTIONS for performing some task, chunked together and given a name. "Chunking" allows you to deal with a potentially very complicated task as a single concept. Instead of worrying about the many, many steps that the computer might have to go through 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 VCR, your refrigerator... You can turn your television on and off, change channels, and set the volume by using elements of the television's interface -- dials, remote control, don't forget to plug in the power -- without understanding anything about how the thing actually works. The same goes for a VCR, although if stories about how hard people find it to set the time on a VCR are true, maybe the VCR violates the simple interface rule.

Now, a black box does have an inside -- the code in a subroutine that actually performs the task, all the electronics inside your television set. The inside of a black box is called its **implementation**. The second rule of black boxes is that

To use a black box, you shouldn't need to know anything about its implementation; all you need to know is its interface.

In fact, it should be possible to change the implementation, as long as the behavior of the box, as seen from the outside, remains unchanged. For example, when the insides of TV sets went from using vacuum tubes to using transistors, the users of the sets didn't even need to know about it -- or even know what it means. Similarly, it should be possible to rewrite the inside of a subroutine, to use more efficient code, for example, without affecting the programs that use that subroutine.

Of course, to have a black box, someone must have designed and built the implementation in the first place. The black box idea works to the advantage of the implementor as well as of 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 6](#), where I will discuss subroutines as a tool in program development.

You should keep in mind that subroutines are not the only example of black boxes in programming. For example, a class is also a black box. We'll see that a class can have a "public" part, representing its interface, and a "private" part that is entirely inside its hidden implementation. All the principles of black boxes apply to classes as well as to subroutines.

[[Next Section](#) | [Previous Chapter](#) | [Chapter Index](#) | [Main Index](#)]

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 scattered all over the Internet. The fact that those subroutines are grouped into named classes (and classes are grouped into named "packages") helps control the confusion that might result from so many different names.

A subroutine that is a member of a class is often called a **method**, and "method" is the term that most people prefer for subroutines in Java. I will start using the term "method" occasionally; however, I will continue to prefer the term "subroutine" for static subroutines. I will use the term "method" most often to refer to non-static subroutines, which belong to objects rather than to classes. This chapter will deal with static subroutines almost exclusively. We'll turn to non-static methods and object-oriented programming in the [next chapter](#).

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 `paint()` routine of an applet. So you are familiar with the general format.

The **statements** between the braces, { and }, 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 method, such as whether it is static or not. The modifiers that you've seen so far are "static" and "public". There are only about a half-dozen possible modifiers altogether.

If the subroutine is a function, whose job is to compute some value, then the **return-type** is used to specify the type of value that is returned by the function. We'll be looking at functions and return types in some detail in [Section 4](#). If the subroutine is not a function, then the **return-type** is replaced by the special value `void`, which indicates that no value is returned. The term "void" is meant to indicate that the return value is empty or non-existent.

Finally, we come to the **parameter-list** of the method. Parameters are part of the interface of a subroutine. They represent information that is passed into the subroutine from outside, to be used by the subroutine's internal computations. For a concrete example, imagine a class named `Television` that includes a method named `changeChannel()`. The immediate question is: What channel should it change to? A parameter can be used to answer this question. Since the channel number is an integer, the type of the parameter would be `int`, and the declaration of the `changeChannel()` method might look like

```
public void changeChannel(int channelNum) {...}
```

This declaration specifies that `changeChannel()` has a parameter named `channelNum` of type `int`. However, `channelNum` does not yet have any particular value. A value for `channelNum` is provided when the subroutine is called; for example: `changeChannel(17);`

The parameter list in a subroutine can be empty, or it can consist of one or more parameter declarations of the form **type parameter-name**. If there are several declarations, they are separated by commas. Note that each declaration can name only one parameter. For example, if you want two parameters of type `double`, you have to say "`double x, double y`", rather than "`double x, y`".

Parameters are covered in more detail in the [next section](#).

Here are a few examples of subroutine definitions, leaving out the statements that define what the subroutines do:

```
public static void playGame() {
    // "public" and "static" are modifiers; "void" is the
    // return-type; "playGame" is the subroutine-name;
    // the parameter-list is empty
    . . . // statements that define what playGame does go here
}

int getNextN(int N) {
    // there are no modifiers; "int" in the return-type
    // "getNextN" is the subroutine-name; the parameter-list
    // includes one parameter whose name is "N" and whose
    // type is "int"
    . . . // statements that define what getNextN does go here
}

static boolean lessThan(double x, double y) {
    // "static" is a modifier; "boolean" is the
    // return-type; "lessThan" is the subroutine-name; the
    // parameter-list includes two parameters whose names are
    // "x" and "y", and the type of each of these parameters
    // is "double"
    . . . // statements that define what lessThan does go here
}
```

In the second example given here, `getNextN`, is a non-static method, since its definition does not include the modifier "`static`" -- and so it's not an example that we should be looking at in this chapter! The other modifier shown in the examples is "`public`". This modifier indicates that the method can be called from anywhere in a program, even from outside the class where the method is defined. There is another modifier, "`private`", which indicates that the method can be called only from inside the same class. The modifiers `public` and `private` are called **access specifiers**. If no access specifier is given for a method, then by default, that method can be called from anywhere in the "package" that contains the class, but not from outside that package. (Packages were mentioned in [Section 3.7](#), and you'll learn more about packages in this chapter, in [Section 5](#).) There is one other access modifier, `protected`, which will only become relevant when we turn to object-oriented programming in [Chapter 5](#).

Note, by the way, that the `main()` routine of a program follows the usual syntax rules for a subroutine. In

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

the modifiers are `public` and `static`, the return type is `void`, the subroutine name is `main`, and the parameter list is "`String[] args`". The only question might be about "`String[]`", which has to be a type if it is to match the format of a parameter list. In fact, `String[]` represents a so-called "array type", so

the syntax is valid. We will cover arrays in [Chapter 8](#). (The parameter, `args`, represents information provided to the program when the `main()` routine is called by the system. In case you know the term, the information consists of any "command-line arguments" specified in the command that the user typed to run the program.)

You've already had some experience with filling in the statements of a subroutine. In this chapter, you'll learn all about writing your own complete subroutine definitions, including the interface part.

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 defined 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. 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** 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 a static subroutine defined elsewhere, in a different class. (Non-static methods belong to objects rather than classes, and they are called using object names instead of class names. More on that later.) Note that the parameter list can be empty, as in the `playGame()` example, but the parentheses must be there even if there is nothing between them.

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 program:

```
Pick a random number
while the game is not over:
    Get the user's guess
```


Tell the user whether the guess is high, low, or correct.

The test for whether the game is over is complicated, since the game ends if either the user makes a correct guess or the number of guesses is six. As in many cases, the easiest thing to do is to use a "while (true)" loop and use break to end the loop whenever we find a reason to do so. Also, if we are going to end the game after six guesses, we'll have to keep track of the number of guesses that the user has made.

Filling out the algorithm gives:

```

Let computersNumber be a random number between 1 and 100
Let guessCount = 0
while (true):
    Get the user's guess
    Count the guess by adding 1 to guess count
    if the user's guess equals computersNumber:
        Tell the user he won
        break out of the loop
    if the number of guesses is 6:
        Tell the user he lost
        break out of the loop
    if the user's guess is less than computersNumber:
        Tell the user the guess was low
    else if the user's guess is higher than computersNumber:
        Tell the user the guess was high

```

With variable declarations added and translated into Java, this becomes the definition of the playGame() routine. A random integer between 1 and 100 can be computed as (int)(100 * Math.random()) + 1. I've cleaned up the interaction with the user to make it flow better.

```

static void playGame() {
    int computersNumber; // A random number picked by the computer.
    int usersGuess;      // A number entered by user as a guess.
    int guessCount;      // Number of guesses the user has made.
    computersNumber = (int)(100 * Math.random()) + 1;
        // The value assigned to computersNumber is a randomly
        // chosen integer between 1 and 100, inclusive.
    guessCount = 0;
    TextIO.putln();
    TextIO.put("What is your first guess? ");
    while (true) {
        usersGuess = TextIO.getInt(); // get the user's guess
        guessCount++;
        if (usersGuess == computersNumber) {
            TextIO.putln("You got it in " + guessCount
                + " guesses! My number was " + computersNumber);
            break; // the game is over; the user has won
        }
        if (guessCount == 6) {
            TextIO.putln("You didn't get the number in 6 guesses.");
            TextIO.putln("You lose. My number was " + computersNumber);
            break; // the game is over; the user has lost
        }
        // If we get to this point, the game continues.
        // Tell the user if the guess was too high or too low.
        if (usersGuess < computersNumber)
            TextIO.put("That's too low. Try again: ");
        else if (usersGuess > computersNumber)
            TextIO.put("That's too high. Try again: ");
    }
}

```

```

    }
    TextIO.putln();
} // end of playGame()

```

Now, where exactly should you put this? It should be part of the same class as the `main()` routine, but not inside the main routine. It is not legal to have one subroutine physically nested inside another. The `main()` routine will call `playGame()`, but not contain it physically. You can put the definition of `playGame()` either before or after the `main()` routine. Java is not very picky about having the members of a class in any particular order.

It's pretty easy to write the main routine. You've done things like this before. Here's what the complete program looks like (except that a serious program needs more comments than I've included here).

```

public class GuessingGame {

    public static void main(String[] args) {
        TextIO.putln("Let's play a game.  I'll pick a number between");
        TextIO.putln("1 and 100, and you try to guess it.");
        boolean playAgain;
        do {
            playGame(); // call subroutine to play one game
            TextIO.put("Would you like to play again? ");
            playAgain = TextIO.getlnBoolean();
        } while (playAgain);
        TextIO.putln("Thanks for playing.  Goodbye.");
    } // end of main()

    static void playGame() {
        int computersNumber; // A random number picked by the computer.
        int usersGuess;      // A number entered by user as a guess.
        int guessCount;      // Number of guesses the user has made.
        computersNumber = (int)(100 * Math.random()) + 1;
        // The value assigned to computersNumber is a randomly
        // chosen integer between 1 and 100, inclusive.
        guessCount = 0;
        TextIO.putln();
        TextIO.put("What is your first guess? ");
        while (true) {
            usersGuess = TextIO.getInt(); // get the user's guess
            guessCount++;
            if (usersGuess == computersNumber) {
                TextIO.putln("You got it in " + guessCount
                    + " guesses!  My number was " + computersNumber);
                break; // the game is over; the user has won
            }
            if (guessCount == 6) {
                TextIO.putln("You didn't get the number in 6 guesses.");
                TextIO.putln("You lose.  My number was " + computersNumber);
                break; // the game is over; the user has lost
            }
            // If we get to this point, the game continues.
            // Tell the user if the guess was too high or too low.
            if (usersGuess < computersNumber)
                TextIO.put("That's too low.  Try again: ");
            else if (usersGuess > computersNumber)
                TextIO.put("That's too high.  Try again: ");
        }
    }
}

```

```

    }
    TextIO.putln();
} // end of playGame()

} // end of class GuessingGame

```

Take some time to read the program carefully and figure out how it works. And try to convince yourself that even in this relatively simple case, breaking up the program into two methods makes the program easier to understand and probably made it easier to write each piece.

You can try out a simulation of this program here:

(Applet "GuessingGameConsole" would be displayed here
if Java were available.)

A class can include other things besides subroutines. In particular, it can also include variable declarations. Of course, you can have variable declarations inside subroutines. Those are called **local variables**. However, you can also have variables that are not part of any subroutine. To distinguish such variables from local variables, we call them **member variables**, since they are members of a class.

Just as with subroutines, member variables can be either static or non-static. In this chapter, we'll stick to static variables. A static member variable belongs to the class itself, and it exists as long as the class exists. Memory is allocated for the variable when the class is first loaded by the Java interpreter. Any assignment statement that assigns a value to the variable changes the content of that memory, no matter where that assignment statement is located in the program. Any time the variable is used in an expression, the value is fetched from that same memory, no matter where the expression is located in the program. This means that the value of a static member variable can be set in one subroutine and used in another subroutine. Static member variables are "shared" by all the static subroutines in the class. A local variable in a subroutine, on the other hand, exists only while that subroutine is being executed, and is completely inaccessible from outside that one subroutine.

The declaration of a member variable looks just like the declaration of a local variable except for two things: The member variable is declared outside any subroutine (although it still has to be inside a class), and the declaration can be marked with modifiers such as `static`, `public`, and `private`. Since we are only working with static member variables for now, every declaration of a member variable in this chapter will include the modifier `static`. For example:

```

static int numberOfPlayers;
static String userName;
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`. If `numberOfPlayers` is a public static member variable in a class named `Poker`, subroutines in the `Poker` class would refer to it simply as `numberOfPlayers`. Subroutines in another class would refer to it as `Poker.numberOfPlayers`.

As an example, let's add a static member variable to the `GuessingGame` class that we wrote earlier in this section. This variable will be used to keep track of how many games the user wins. We'll call the variable `gamesWon` and declare it with the statement `static int gamesWon;` In the `playGame()` routine, we add 1 to `gamesWon` if the user wins the game. At the end of the `main()` routine, we print out the value of `gamesWon`. It would be impossible to do the same thing with a local variable, since we need access to the same variable from both subroutines.

When you declare a local variable in a subroutine, you have to assign a value to that variable before you can do anything with it. Member variables, on the other hand are automatically initialized with a default value. For numeric variables, the default value is zero. For boolean variables, the default is false. And for char variables, it's the unprintable character that has Unicode code number zero. (For objects, such as Strings, the default initial value is a special value called null, which we won't encounter officially until later.)

Since it is of type `int`, the static member variable `gamesWon` automatically gets assigned an initial value of zero. This happens to be the correct initial value for a variable that is being used as a counter. You can, of course, assign a different value to the variable at the beginning of the `main()` routine if you are not satisfied with the default initial value.

Here's a revised version of `GuessingGame.java` that includes the `gamesWon` variable. The changes from the above version are shown in red:

```
public class GuessingGame2 {

    static int gamesWon;          // The number of games won by
                                // the user.

    public static void main(String[] args) {
        gamesWon = 0; // This is actually redundant, since 0 is
                       // the default initial value.
        TextIO.putln("Let's play a game. I'll pick a number between");
        TextIO.putln("1 and 100, and you try to guess it.");
        boolean playAgain;
        do {
            playGame(); // call subroutine to play one game
            TextIO.put("Would you like to play again? ");
            playAgain = TextIO.getlnBoolean();
        } while (playAgain);
        TextIO.putln();
        TextIO.putln("You won " + gamesWon + " games.");
        TextIO.putln("Thanks for playing. Goodbye.");
    } // end of main()

    static void playGame() {
        int computersNumber; // A random number picked by the computer.
        int usersGuess;      // A number entered by user as a guess.
        int guessCount;      // Number of guesses the user has made.
        computersNumber = (int)(100 * Math.random()) + 1;
        // The value assigned to computersNumber is a randomly
        // chosen integer between 1 and 100, inclusive.
        guessCount = 0;
        TextIO.putln();
        TextIO.put("What is your first guess? ");
        while (true) {
            usersGuess = TextIO.getInt(); // get the user's guess
            guessCount++;
            if (usersGuess == computersNumber) {
                TextIO.putln("You got it in " + guessCount
                    + " guesses! My number was " + computersNumber);
                gamesWon++; // Count this game by incrementing gamesWon.
                break;      // the game is over; the user has won
            }
        }
    }
}
```

```
        if (guessCount == 6) {
            TextIO.putln("You didn't get the number in 6 guesses.");
            TextIO.putln("You lose.  My number was " + computersNumber);
            break; // the game is over; the user has lost
        }
        // If we get to this point, the game continues.
        // Tell the user if the guess was too high or too low.
        if (usersGuess < computersNumber)
            TextIO.put("That's too low.  Try again: ");
        else if (usersGuess > computersNumber)
            TextIO.put("That's too high.  Try again: ");
    }
    TextIO.putln();
} // end of playGame()

} // end of class GuessingGame2
```

[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Section 4.3

Parameters

IF A SUBROUTINE IS A BLACK BOX, then a parameter 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.

As an example, let's go back to the "3N+1" problem that was discussed in [Section 3.2](#). (Recall that a 3N+1 sequence is computed according to the rule, "if N is odd, multiply by 3 and add 1; if N is even, divide by 2; continue until N is equal to 1." For example, starting from N=3 we get the sequence: 3, 10, 5, 16, 8, 4, 2, 1.) Suppose that we want to write a subroutine to print out such sequences. The subroutine will always perform the same task: Print out a 3N+1 sequence. But the exact sequence it prints out depends on the starting value of N. So, the starting value of N would be a parameter to the subroutine. The subroutine could be written like this:

```
static void Print3NSequence(int startingValue) {

    // 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.

    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.

    int count = 1; // We have one term, the starting value, so far.

    TextIO.putln("The 3N+1 sequence starting from " + N);
    TextIO.putln();
    TextIO.putln(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
        TextIO.putln(N); // print this term
    }

    TextIO.putln();
    TextIO.putln("There were " + count + " terms in the sequence.");
}
```

```
    } // end of Print3NSequence()
```

The parameter list of this subroutine, "(int startingValue)", specifies that the subroutine has one parameter, of type `int`. When the subroutine is called, a value must be provided for this parameter. This value is assigned to the parameter, `startingValue`, before the body of the subroutine is executed. For example, the subroutine could be called using the subroutine call statement `"Print3NSequence(17);"`. When the computer executes this statement, the computer assigns the value 17 to `startingValue` and then executes the statements in the subroutine. This prints the $3N+1$ sequence starting from 17. If `K` is a variable of type `int`, then when the computer executes the subroutine call statement `"Print3NSequence(K);"`, it will take the value of the variable `K`, assign that value to `startingValue`, and execute the body of the subroutine.

The class that contains `Print3NSequence` can contain a `main()` routine (or other subroutines) that call `Print3NSequence`. For example, here is a `main()` program that prints out $3N+1$ sequences for various starting values specified by the user:

```
public static void main(String[] args) {
    TextIO.putln("This program will print out 3N+1 sequences");
    TextIO.putln("for starting values that you specify.");
    TextIO.putln();
    int K; // Input from user; loop ends when K < 0.
    do {
        TextIO.putln("Enter a starting value;")
        TextIO.put("To end the program, enter 0: ");
        K = TextIO.getInt(); // get starting value from user
        if (K > 0) // print sequence, but only if K is > 0
            Print3NSequence(K);
    } while (K > 0); // continue only if K > 0
} // end main()
```

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**. 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 an identifier, that is, a name. A formal parameter is very much like a variable, and -- like a variable -- it has a specified type such as `int`, `boolean`, or `String`. An actual parameter is a value, and so it can be specified by any expression, provided that the expression computes a value of the correct type. The type of the actual parameter must be one that could legally be assigned to the formal parameter with an assignment statement. For example, if the formal parameter is of type `double`, then it would be legal to pass an `int` as the actual parameter since `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 mistake is to assign values to the formal parameters at the beginning of the subroutine, or to ask the user to input their values. This represents a fundamental misunderstanding. When the statements in the subroutine are executed, the formal parameters will already have values. The values come from the subroutine call statement. Remember that a subroutine is not independent. It is called by some other routine, and it is the calling routine's responsibility to provide appropriate values for the parameters.

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` can be expressed as `doTask(int, double, boolean)`. Note that the signature does not include the names of the parameters; in fact, if you just want to use the subroutine, you don't even need to know what the formal parameter names are, so the names are not part of the interface.

Java is somewhat unusual in that it allows two different subroutines in the same class to have the same name, provided that their signatures are different. (The language C++ on which Java is based also has this feature.) 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 in the `TextIO` class. This class includes many different methods named `putln`, for example. These methods all have different signatures, such as:

```
putln(int)          putln(int, int)          putln(double)
putln(String)       putln(String, int)       putln(char)
putln(boolean)      putln(boolean, int)      putln()
```

Of course all these different subroutines are semantically related, which is why it is acceptable programming style to use the same name for them all. But as far as the computer is concerned, printing out an `int` is very different from printing out a `String`, which is different from printing out a `boolean`, and so forth -- so that each of these operations requires a different method.

Note, by the way, that the signature does not include the subroutine's return type. It is illegal to have two subroutines in the same class that have the same signature but that have different return types. For example, it would be a syntax error for a class to contain two methods defined as:

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

So it should be no surprise that in the `TextIO` class, the methods for reading different types are not all named `getln()`. In a given class, there can only be one routine that has the name `getln` and has no parameters. The input routines in `TextIO` are distinguished by having different names, such as `getlnInt()` and `getlnDouble()`.

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 6](#).

As a first example, let's write a subroutine to compute and print out all the divisors of a given positive integer. The integer will be a parameter to the subroutine.

Remember that the format of any subroutine is

```
modifiers return-type subroutine-name ( parameter-list ) {
    statements
}
```

Writing a subroutine always means filling out this format. The assignment tells us that there is one parameter, of type `int`, and it tells us what the statements in the body of the subroutine should do. Since we are only working with static subroutines for now, we'll need to use `static` as a modifier. We could add an access modifier (`public` or `private`), but in the absence of any instructions, I'll leave it out. Since we are not told to return a value, the return type is `void`. Since no names are specified, we'll have to make up names for the formal parameter and for the subroutine itself. I'll use `N` for the parameter and `printDivisors` for the subroutine name. The subroutine will look like

```
static void printDivisors( int N ) {
    statements
}
```

and all we have left to do is to write the statements that make up the body of the routine. This is not difficult. Just remember that you have to write the body assuming that `N` already has a value! The algorithm is: "For each possible divisor `D` in the range from 1 to `N`, if `D` evenly divides `N`, then print `D`." Written in Java, this becomes:

```
static void printDivisors( int N ) {
    // Print all the divisors of N.
    // We assume that N is a positive integer.
    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 )
            System.out.println(D);
    }
}
```

I've added comments 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 assignment: Write a subroutine named `printRow`. It should have a parameter `ch` of type `char` and a parameter `N` of type `int`. The subroutine should print out a line of text containing `N` copies of the character `ch`.

Here, we are told the name of the subroutine and the names of the two parameters, so we don't have much choice about the first line of the subroutine definition. The task in this case is pretty simple, so the body of the subroutine is easy to write. The complete subroutine is given by

```
static void printRow( char ch, int N ) {
    // Write one line of output containing N copies of the
    // character ch.  If N <= 0, an empty line is output.
    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 will print a line of output containing 25 copies of that character. It should use the `printRow()` subroutine to produce the output.

Again, we get to choose a name for the subroutine and a name for the parameter. I'll call the subroutine `printRowsFromString` and the parameter `str`. The algorithm is pretty clear: For each position i in the string `str`, call `printRow(str.charAt(i), 25)` to print one line of the output. So, we get:

```
static void printRowsFromString( String str ) {
    // For each character in str, write a line of output
    // containing 25 copies of that character.
    int i; // Loop-control variable for counting off the chars.
    for ( i = 0; i < str.length(); i++ ) {
        printRow( str.charAt(i), 25 );
    }
}
```

We could use `printRowsFromString` in a `main()` routine such as

```
public static void main(String[] args) {
    String inputLine; // Line of text input by user.
    TextIO.put("Enter a line of text: ");
    inputLine = TextIO.getln();
    TextIO.putln();
    printRowsFromString( inputLine );
}
```

Of course, the three routines, `main()`, `printRowsFromString()`, and `printRow()`, would have to be collected together inside the same class. The program is rather useless, but it does demonstrate the use of subroutines. You'll find the program in the file [RowsOfChars.java](#), if you want to take a look. Here's an applet that simulates the program:

(Applet "RowsOfCharsConsole" would be displayed here
if Java were available.)

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 defined in the subroutine, formal parameter names, and static member variables that are defined outside the subroutine but inside the same class as the subroutine.

Local variables have no connection to the outside world; they are purely part of the internal working of the

subroutine. Parameters are used to "drop" values into the subroutine when it is called, but once the subroutine starts executing, parameters act much like local variables. Changes made inside a subroutine to a formal parameter have no effect on the rest of the program (at least if the type of the parameter is one of the primitive types -- things are more complicated in the case of objects, as we'll see later).

Things are different when a subroutine uses a variable that is defined outside the subroutine. That variable exists independently of the subroutine, and it is accessible to other parts of the program, as well as to the subroutine. Such a variable is said to be **global** to the subroutine, as opposed to the "local" variables defined inside the subroutine. The scope of a global variable includes the entire class in which it is defined. Changes made to a global variable can have effects that extend outside the subroutine where the changes are made. You've seen how this works in the last example in the [previous section](#), where the value of the global variable, `gamesWon`, is computed inside a subroutine and is used in the `main()` routine.

It's not always bad to use global variables in subroutines, but you should realize that the global variable then has to be considered part of the subroutine's interface. The subroutine uses the global variable to communicate with the rest of the program. This is a kind of sneaky, back-door communication that is less visible than communication done through parameters, and it risks violating the rule that the interface of a black box should be straightforward and easy to understand. So before you use a global variable in a subroutine, you should consider whether it's really necessary.

I don't advise you to take an absolute stand against using global variables inside subroutines. There is at least one good reason to do it: If you think of the class as a whole as being a kind of black box, it can be very reasonable to let the subroutines inside that box be a little sneaky about communicating with each other, if that will make the class as a whole look simpler from the outside.

[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

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`, 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 this input is not assigned to a variable or used in an expression, it is simply discarded. Sometimes, discarding unwanted input is exactly what you need to do.)

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 takes the form:

`return expression;`

Such a **return statement** can only occur inside the definition of a function, and the type of the **expression** must match the return type that was specified for the function. (More exactly, it must be legal to assign the expression to a variable whose type is specified by the return type.) When the computer executes this **return statement**, it evaluates the expression, terminates execution of the function, and uses the value of the expression as the returned value of the function.

For example, consider the function definition

```
static double pythagorus(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 + pythagorus(12,5);`. When it gets to the term `pythagorus(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, so it replaces the function call in the statement `totalLength = 17 + pythagorus(12,5);`. The return value is added to 17, and the result, 30.0, is stored in the variable, `totalLength`. The effect is the same as if the statement had been `totalLength = 17 + 13.0;`.

Inside an ordinary subroutine -- with declared return type `"void"` -- you can use a **return statement** with no expression to immediately 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.

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.) 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
}
```

Exactly one of the two return statements is executed to give the value of the function. A return statement can occur anywhere in a function. Some people, however, prefer to use a single return statement at the very end of the function. 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 in [Section 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) {

    // 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 startingValue must be a positive integer.

    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;

    TextIO.putln("The 3N+1 sequence starting from " + N);
    TextIO.putln();
    TextIO.putln(N); // print initial term of sequence

    while (N > 1) {
        N = nextN( N ); // Compute next term,
                        // using the function nextN.
        count++;        // Count this term.
        TextIO.putln(N); // Print this term.
    }

    TextIO.putln();
    TextIO.putln("There were " + count + " terms in the sequence.");

} // end of Print3NSequence()
```

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:

```
static char letterGrade(int numGrade) {

    // Returns the letter grade corresponding to
    // the numerical grade, 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:

```
static boolean isPrime(int N) {

    // 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).

    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 = (int)Math.sqrt(N);
        // We will try to divide N by numbers between
        // 2 and maxToTry; If N is not evenly divisible
        // by any of these numbers, then N is prime.
        // (Note that since Math.sqrt(N) is defined to
        // return a value of type double, the value
        // must be typecast to type int before it can
        // be assigned to maxToTry.)

    for (divisor = 2; divisor <= maxToTry; divisor++) {
        if ( N % divisor == 0 )    // Test if divisor evenly divides N.
            return false;        // If so, we know N is not prime.
                                   // No need to continue testing.
    }

    // If we get to this point, N must be prime.  Otherwise,
```



```

        // the function would already have been terminated by
        // a return statement in the previous for 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) {
    // Returns a reversed copy of 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.

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. This 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 columns, I will use the version of `TextIO.put()` with signature `put(int,int)`. The second `int` parameter tells how wide the columns should be.

```

public class ThreeN2 {

    /*
       A program that computes and displays several 3N+1
       sequences. Starting values for the sequences are
       input by the user. Terms in a 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 static void main(String[] args) {

        TextIO.putln("This program will print out 3N+1 sequences");
        TextIO.putln("for starting values that you specify.");
        TextIO.putln();

        int K;    // Starting point for sequence, specified by the user.
        do {
            TextIO.putln("Enter a starting value;");
            TextIO.put("To end the program, enter 0: ");
            K = TextIO.getInt();    // get starting value from user
            if (K > 0)                // print sequence, but only if K is > 0
                Print3NSequence(K);
        } while (K > 0);            // continue only if K > 0

    } // end main()

    static void Print3NSequence(int startingValue) {

        // Prints a 3N+1 sequence to standard output, using
        // startingValue as the initial value of N.  Terms are
        // printed five to a line.  The subroutine also
        // prints the number of terms in the sequence.
        // The value of startingValue must be a positive integer.

        int N;        // One of the terms in the sequence.
        int count;    // The number of terms found.
        int onLine;   // The number of terms that have been output
                     // so far on the current line.

        N = startingValue;    // Start the sequence with startingValue;
        count = 1;            // We have one term so far.

        TextIO.putln("The 3N+1 sequence starting from " + N);
        TextIO.putln();
        TextIO.put(N, 8);    // Print initial term, using 8 characters.
        onLine = 1;         // There's now 1 term on current output line.

        while (N > 1) {
            N = nextN(N);    // compute next term
            count++;        // count this term
            if (onLine == 5) { // If current output line is full
                TextIO.putln(); // ...then output a carriage return
                onLine = 0;    // ...and note that there are no terms
                             // on the new line.
            }
            TextIO.put(N, 8); // Print this term in an 8-char column.
            onLine++;        // Add 1 to the number of terms on this line.
        }

        TextIO.putln();    // end current line of output
        TextIO.putln();    // and then add a blank line
        TextIO.putln("There were " + count + " terms in the sequence.");
    }

```

```
    } // end of Print3NSequence()

    static int nextN(int currentN) {
        // Computes and returns the next term in a 3N+1 sequence,
        // given that the current term is 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. Here is an applet version for you to try:

**(Applet "ThreeN2Console" would be displayed here
if Java were available.)**

[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Section 4.5

Toolboxes, API's, and Packages

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.

Someone who wants to program for Macintosh computers -- and to produce programs that look and behave the way users expect them to -- must deal with the Macintosh Toolbox, a collection of well over a thousand different subroutines. There are routines for opening and closing windows, for drawing geometric figures and text to windows, for adding buttons to windows, and for responding to mouse clicks on the window. There are other routines for creating menus and for reacting to user selections from menus. Aside from the user interface, there are routines for opening files and reading data from them, for communicating over a network, for sending output to a printer, for handling communication between programs, and in general for doing all the standard things that a computer has to do. Windows 98 and Windows 2000 provide their own sets of subroutines for programmers to use, and they are quite a bit different from the subroutines used on the Mac.

The analogy of a "toolbox" is a good one to keep in mind. Every programming project involves a mixture of innovation and reuse of existing tools. A programmer is given a set of tools to work with, starting with the set of basic tools that are built into the language: things like variables, assignment statements, if statements, and loops. To these, the programmer can add existing toolboxes full of routines that have already been written for performing certain tasks. These tools, if they are well-designed, can be used as true black boxes: They can be called to perform their assigned tasks without worrying about the particular steps they go through to accomplish those tasks. The innovative part of programming is to take all these tools and apply them to some particular project or problem (word-processing, keeping track of bank accounts, processing image data from a space probe, Web browsing, computer games,...). This is called **applications programming**.

A software toolbox is a kind of black box, and it presents a certain interface to the programmer. This interface is a specification of what routines are in the toolbox, what parameters they use, and what tasks they perform. This information constitutes the API, or **Applications Programming Interface**, associated with the toolbox. The Macintosh API is a specification of all the routines available in the Macintosh Toolbox. A company that makes some hardware device -- say a card for connecting a computer to a network -- might publish an API for that device consisting of a list of routines that programmers can call in order to communicate with and control the device. Scientists who write a set of routines for doing some kind of complex computation -- such as solving "differential equations", say -- would provide an API to allow others to use those routines without understanding the details of the computations they perform.

The Java programming language is supplemented by a large, standard API. You've seen part of this API already, in the form of mathematical subroutines such as `Math.sqrt()`, the `String` data type and its associated routines, and the `System.out.print()` routines. The standard Java API includes routines for working with graphical user interfaces, for network communication, for reading and writing files, and more. It's tempting to think of these routines as being built into the Java language, but they are technically subroutines that have been written and made available for use in Java programs.

Java is platform-independent. That is, the same program can run on platforms as diverse as Macintosh, Windows, UNIX, and others. The same Java API must work on all these platforms. But notice that it is the

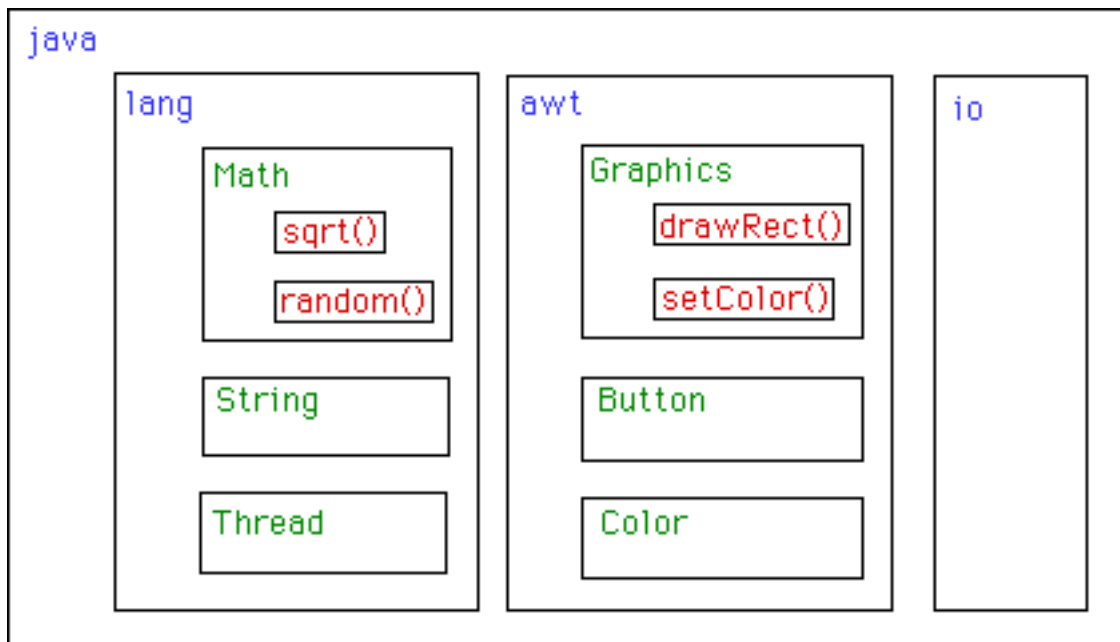
interface that is platform-independent; the implementation varies from one platform to another. A Java system on a particular computer includes implementations of all the standard API routines. A Java program includes only calls to those routines. When the Java interpreter executes a program and encounters a call to one of the standard routines, it will pull up and execute the implementation of that routine which is appropriate for the particular platform on which it is running. This is a very powerful idea. It means that you only need to learn one API to program for a wide variety of platforms.

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**. 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 the non-GUI packages as well as the original AWT graphics user interface classes. Another package, "javax", was added in Java version 1.2 and contains the classes used by the Swing graphical user interface.

A package can contain both classes and other packages. A package that is contained in another package is sometimes called a "sub-package." Both the java package and the javax package contain sub-packages. One of the sub-packages of java, for example, is called "awt". Since awt is contained within java, its full name is actually java.awt. This is the package that contains classes related to the AWT graphical user interface, such as a Button class which represents push-buttons on the screen and the Graphics class which provides routines for drawing on the screen. Since these classes are contained in the package java.awt, their full names are actually java.awt.Button and java.awt.Graphics. (I hope that by now you've gotten the hang of how this naming thing works in Java.) Similarly, javax contains a sub-package named javax.swing, which includes such classes as javax.swing.JButton and javax.swing.JApplet.

The java package includes several other sub-packages, such as java.io, which provides facilities for input/output, java.net, which deals with network communication, and java.applet, which implements the basic functionality of applets. The most basic package is called java.lang. This package contains fundamental classes such as String and Math.

It might be helpful to look at a graphical representation of the levels of nesting in the java package, its sub-packages, the classes in those sub-packages, and the subroutines in those classes. This is not a complete picture, since it shows only a 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()

Let's say that you want to use the class `java.awt.Color` in a program that you are writing. One way to do this is to use the full name of the class. For example, you could say

```
java.awt.Color rectColor;
```

to declare a variable named `rectColor` whose type is `java.awt.Color`. Of course, this can get tiresome, so Java makes it possible to avoid using the full names of classes. If you put

```
import java.awt.Color;
```

at the beginning of a Java source code file, then, in the rest of the file, you can abbreviate the full name `java.awt.Color` to just the name of the class, `Color`. This would allow you to say just

```
Color rectColor;
```

to declare the variable `rectColor`. (The only effect of the `import` statement 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` statement, you can still access the class -- you just have to use its full name.) There is a shortcut for importing all the classes from a given package. You can import all the classes from `java.awt` by saying

```
import java.awt.*;
```

and you can import all the classes from `javax.swing` with the line

```
import javax.swing.*;
```

In fact, any Java program that uses a graphical user interface is likely to begin with one or both of these lines. A program might also include lines such as `"import java.net.*;"` or `"import java.io.*;"` to get easy access to networking and input/output classes. (When you start importing lots of packages in this way, you have to be careful about one thing: It's possible for two classes that are in different packages to have the same name. For example, both the `java.awt` package and the `java.util` package contain classes named `List`. If you import both `java.awt.*` and

`java.util.*`, the simple name `List` will be ambiguous. If you try to declare a variable of type `List`, you will get a compiler error message about an ambiguous class name. The solution is simple: use the full name of the class, either `java.awt.List` or `java.util.List`. Another solution is to use `import` to import the individual classes you need, instead of importing entire packages.)

Because the package `java.lang` is so fundamental, all the classes in `java.lang` are automatically imported into every program. It's as if every program began with the statement `"import java.lang.*;"`. This is why we have been able to use the class name `String` instead of `java.lang.String`, and `Math.sqrt()` instead of `java.lang.Math.sqrt()`. It would still, however, be perfectly legal to use the longer forms of the names.

Programmers can create new packages. Suppose that you want some classes that you are writing to be in a package named `utilities`. Then the source code file that defines those classes must begin with the line `"package utilities;"`. Any program that uses the classes should include the directive `"import utilities.*;"` to obtain access to all the classes in the `utilities` package. Unfortunately, things are a little more complicated than this. Remember that if a program uses a class, then the class must be "available" when the program is compiled and when it is executed. Exactly what this means depends on which Java environment you are using. Most commonly, classes in a package named `utilities` should be in a directory with the name `utilities`, and that directory should be located in the same place as the program that uses the classes.

In projects that define large numbers of classes, it makes sense to organize those classes into one or more packages. It also makes sense for programmers to create new packages as toolboxes that provide functionality and API's 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, I will not be creating any packages in this textbook. You need to know about packages mainly so that you will be able to import the standard packages. These packages are always available to the programs that you write. You might wonder where the standard classes are actually located. Again, that depends to some extent on the version of Java that you are using. But they are likely to be collected together into a large file named `rt.jar` or `classes.zip`, which is located in some place where the Java compiler and the Java interpreter will know to look for it.

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. All the examples that you see in these notes are in the default package.

[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Section 4.6

More on Program Design

UNDERSTANDING HOW PROGRAMS WORK IS ONE THING. Designing a program to perform some particular task is another thing altogether. In [Section 3.2](#), I discussed how 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 programming language. 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.

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 1](#). A convenient way to express the contract of a subroutine is in terms of **preconditions** and **postconditions**.

The precondition of a subroutine is something that must be true when the subroutine is called, if the subroutine is to work correctly. For example, for the built-in function `Math.sqrt(x)`, a precondition is that the parameter, `x`, is greater than or equal to zero, since it is not possible to take the square root of a negative number. In terms of a contract, a precondition represents an obligation of the *caller* of the subroutine. **If you call a subroutine without meeting its precondition, then there is no reason to expect it to work properly. The program might crash or give incorrect results, but you can only blame yourself, not the subroutine.**

A postcondition of a subroutine represents the other side of the contract. It is something that will be true after the subroutine has run (assuming that its preconditions were met -- and that there are no bugs in the subroutine). The postcondition of the function `Math.sqrt()` is that the square of the value that is returned by this function is equal to the parameter that is provided when the subroutine is called. Of course, this will only be true if the precondition -- that the parameter is greater than or equal to zero -- is met. A postcondition of the built-in subroutine `System.out.print()` is that the value of the parameter has been displayed on the screen.

Preconditions most often give restrictions on the acceptable values of parameters, as in the example of `Math.sqrt(x)`. However, they can also refer to global variables that are used in the subroutine. The postcondition of a subroutine specifies the task that it performs. For a function, the postcondition should specify the value that the function returns.

Subroutines are often 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. I will also use it occasionally later in the book, although I will generally be less formal in my commenting style.

Preconditions and postconditions will be discussed more thoroughly in [Chapter 9](#), which deals with techniques for writing correct and robust programs.

A Design Example

Let's work through an example of program design using subroutines. In this example, we will both use prewritten subroutines as building blocks and design new subroutines that we need to complete the project.

Suppose that I have found an already-written class called `Mosaic`. This class allows a program to work with a window that displays little colored rectangles arranged in rows and columns. The window can be opened, closed, and otherwise manipulated with static member subroutines defined in the `Mosaic` class. Here are some of the available routines:

```
void Mosaic.open(int rows, int cols, int w, int h);
    Precondition:  The parameters rows, cols, w, and h are
                   greater than zero.
    Postcondition: A "mosaic" window is opened 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 the rectangles are black.
    Note: The rows are numbered from 0 to rows - 1, and the columns
          are numbered from 0 to cols - 1.

void Mosaic.setColor(int row, int col, int r, int g, int b);
    Precondition:  row and col are in the valid ranges of row numbers and
                   column numbers.  r, g, and b are in the range 0 to 255.
                   Also, the mosaic window should be open.
    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.

int Mosaic.getRed(int row, int col);
int Mosaic.getBlue(int row, int col);
int Mosaic.getGreen(int row, int col);
    Precondition:  row and col are in the valid ranges of row numbers
                   and column numbers.  Also, the mosaic window should
                   be open.
    Postcondition: Returns an int value that represents one of the
                   three color components of the rectangle in row
                   number row and column number col.  The return value
                   is in the range 0 to 255.  (Mosaic.getRed() returns
                   the red component of the rectangle, Mosaic.getGreen()
                   the green component, and Mosaic.getBlue() the blue
                   component.)

void Mosaic.delay(int milliseconds);
    Precondition:  milliseconds is a positive integer.
    Postcondition: The program has paused for at least the number
```

of milliseconds given by the parameter, where one second is equal to 1000 milliseconds.

Note: This can be used to insert a time delay in the program (to regulate the speed at which the colors are changed, for example).

```
boolean Mosaic.isOpen();
Precondition: None.
Postcondition: The return value is true if the mosaic window
               is open on the screen, and is false otherwise.
Note: The window will be closed if the user clicks its
      close box. It can also be closed programmatically by
      calling the subroutine Mosaic.close().
```

My idea 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 an applet that shows what the program will do:

(Applet "RandomMosaicWalkApplet" would be displayed here
if Java were available.)

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 a new square and changing the color of that square. 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 current square;
    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`. I'll use 10 rows and 20 columns of squares in my mosaic, so setting the current position to be in the center means setting `currentRow` to 5 and `currentColumn` to 10. I already have a subroutine, `Mosaic.open()`, to open the window, and I have a function, `Mosaic.isOpen()`, to test whether the window is open. To keep the main routine simple, I decide that I will write two more subroutines of my own to carry out the two tasks in the while loop. The algorithm can then be written in Java as:

```
Mosaic.open(10,20,10,10)
fillWithRandomColors();
currentRow = 5;           // Middle row, halfway down the window.
currentColumn = 10;       // Middle column.
while ( Mosaic.isOpen() ) {
    changeToRandomColor(currentRow, currentColumn);
    randomMove();
}
```

With the proper wrapper, this is essentially the `main()` routine of my program. It turns out I have to make one small modification: To prevent the animation from running too fast, the line `"Mosaic.delay(20);"` is added to

the while loop.

The `main()` routine is taken care of, but to complete the program, I still have to write the subroutines `fillWithRandomColors()`, `changeToRandomColor(int,int)`, and `randomMove()`. Writing each of these subroutines is a separate, small task. The `fillWithRandomColors()` routine is defined by the postcondition that "each of the rectangles in the mosaic has been changed to a random color." Pseudocode for an algorithm to accomplish this task can be given as:

```
For each row:
    For each column:
        set the square in that row and column to a random color
```

"For each row" and "for each column" can be implemented as for loops. We've already planned to write a subroutine `changeToRandomColor` that can be used to set the color. (The possibility of reusing subroutines in several places is one of the big payoffs of using them!) So, `fillWithRandomColors()` can be written in proper Java as:

```
static void fillWithRandomColors() {
    for (int row = 0; row < 10; row++)
        for (int column = 0; column < 20; column++)
            changeToRandomColor(row,column);
}
```

Turning to the `changeToRandomColor` subroutine, we already have a method, `Mosaic.setColor(row,col,r,g,b)`, 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. Rather than let this happen, I decide to move the disturbance to the opposite edge of the applet by setting `currentRow` to 9. (Remember that the 10 rows are numbered from 0 to 9.) Moving the disturbance down, left, or right is handled similarly. If we use a `switch` statement to decide which direction to move, the code for `randomMove` becomes:

```
int directionNum;
directionNum = (int)(4*Math.random());
switch (directionNum) {
    case 0: // move up
        currentRow--;
        if (currentRow < 0) // CurrentRow is outside the mosaic;
            currentRow = 9; // move it to the opposite edge.
        break;
    case 1: // move right
        currentColumn++;
        if (currentColumn >= 20)
            currentColumn = 0;
        break;
    case 2: // move down
```

```

        currentRow++;
        if (currentRow >= 10)
            currentRow = 0;
        break;
    case 3: // move left
        currentColumn--;
        if (currentColumn < 0)
            currentColumn = 19;
        break;
}

```

Putting this all together, we get the following complete program. The variables `currentRow` and `currentColumn` are defined as static members of the class, rather than local variables, because each of them is used in several different subroutines. This program actually depends on two other classes, `Mosaic` and another class called `MosaicCanvas` that is used by `Mosaic`. If you want to compile and run this program, both of these classes must be available to the program.

```

public class RandomMosaicWalk {

    /*
     * This program shows 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.
     */

    static int currentRow; // row currently containing the disturbance
    static int currentColumn; // column currently containing disturbance

    public static void main(String[] args) {
        // Main program creates the window, fills it with
        // random colors, then moves the disturbance in
        // a random walk around the window for as long as
        // the window is open.
        Mosaic.open(10,20,10,10);
        fillWithRandomColors();
        currentRow = 5; // start at center of window
        currentColumn = 10;
        while (Mosaic.isOpen()) {
            changeToRandomColor(currentRow, currentColumn);
            randomMove();
            Mosaic.delay(20);
        }
    } // end of main()

    static void fillWithRandomColors() {
        // Precondition: The mosaic window is open.
        // Postcondition: Each rectangle has been set to a
        // random color
        for (int row=0; row < 10; row++) {
            for (int column=0; column < 20; column++) {
                changeToRandomColor(row, column);
            }
        }
    } // end of fillWithRandomColors()

    static void changeToRandomColor(int rowNum, int colNum) {
        // Precondition: rowNum and colNum are in the valid range
    }
}

```

```

        // of row and column numbers.
        // Postcondition: The rectangle in the specified row and
        // column has been changed to a random color.
        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 of changeToRandomColor()

    static void randomMove() {
        // Precondition: The global variables currentRow and currentColumn
        // specify a valid position in the grid.
        // Postcondition: currentRow or currentColumn is changed to
        // one of the neighboring positions in the grid,
        // up, down, left, or right from the previous
        // position. If this moves the position outside
        // the grid, then it is moved to the opposite edge
        // of the window.
        int directionNum; // Randomly set to 0, 1, 2, or 3 to choose direction.
        directionNum = (int)(4*Math.random());
        switch (directionNum) {
            case 0: // move up
                currentRow--;
                if (currentRow < 0)
                    currentRow = 9;
                break;
            case 1: // move right
                currentColumn++;
                if (currentColumn >= 20)
                    currentColumn = 0;
                break;
            case 2: // move down
                currentRow ++;
                if (currentRow >= 10)
                    currentRow = 0;
                break;
            case 3: // move left
                currentColumn--;
                if (currentColumn < 0)
                    currentColumn = 19;
                break;
        }
    } // end of randomMove()

} // end of class RandomMosaicWalk

```

[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Section 4.7

The Truth about Declarations

NAMES ARE FUNDAMENTAL TO PROGRAMMING, as I said a few chapters ago. There are a lot of details involved in declaring and using names. I have been avoiding some of those details. In this section, I'll reveal most of the truth (although still not the full truth) about declaring and using variables in Java. The material under the headings "Combining Initialization with Declaration" and "Named Constants and the `final` Modifier" is particularly important, since I will be using it regularly in future chapters.

Combining Initialization with Declaration

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 therefor be abbreviated as

```
int count = 0;  // Declare count and give it an initial value.
```

The computer still executes this statement in two steps: Declare the variable `count`, then assign the value 0 to the newly created variable. The initial value does not have to be a constant. It can be any expression. It is legal to initialize several variables in one declaration statement. For example,

```
char firstInitial = 'D', secondInitial = 'E';

int x, y = 1;      // OK, but only y has been initialized!

int N = 3, M = N+2; // OK, N is initialized
                  //           before its value is used.
```

This feature is especially common in `for` loops, since it makes it possible to declare a loop control variable at the same point in the loop where it is initialized. Since the loop control variable generally has nothing to do with the rest of the program outside the loop, it's reasonable to have its declaration in the part of the program where it's actually used. For example:

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

Again, you should remember that this is simply an abbreviation for the following, where I've added an extra pair of braces to show that `i` is considered to be local to the `for` statement and no longer exists after the `for` loop ends:

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

A member variable can also be initialized at the point where it is declared. For example:

```
public class Bank {
    static double interestRate = 0.05;
    static int maxWithdrawal = 200;
}
```



```

        .    // More variables and subroutines.
        .
    }

```

A static member variable is created as soon as the class is loaded by the Java interpreter, and the initialization is also done at that time. In the case of member variables, this is not simply an abbreviation for a declaration followed by an assignment statement. Declaration statements are the only type of statement that can occur outside of a subroutine. Assignment statements cannot, so the following is illegal:

```

public class Bank {
    static double interestRate;
    interestRate = 0.05;    // ILLEGAL:
    .                      // Can't be outside a subroutine!
    .
    .
}

```

Because of this, declarations of member variables often include initial values. As mentioned in [Section 2](#), if no initial value is provided for a member variable, then a default initial value is used. For example, "static int count;" is equivalent to "static int count = 0;".

Named Constants and the final Modifier

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 that is meant to be the value throughout the entire program. In this case, the programmer is probably defining the variable, `interestRate`, to give a meaningful name to the otherwise meaningless number, 0.05. It's easier to understand what's going on when a program says "principal += principal*interestRate;" rather than "principal += principal*0.05;".

In Java, the modifier "final" can be applied to a variable declaration to ensure that the value of the variable cannot be changed after the variable has been initialized. For example, if the member variable `interestRate` is declared with

```
final static double interestRate = 0.05;
```

then it would be impossible for the value of `interestRate` to change anywhere else in the program. Any assignment statement that tries to assign a value to `interestRate` will be rejected by the computer as a syntax error when the program is compiled.

It is legal to apply the `final` modifier to local variables and even to formal parameters, but it is most useful for member variables. I will often refer to a static member variable that is declared to be `final` as a **named constant**, since its value remains constant for the whole time that the program is running. The readability of a program can be greatly enhanced by using named constants to give meaningful names to important quantities in the program. A recommended style rule for named constants is to give them names that consist entirely of upper case letters, with underscore characters to separate words if necessary. For example, the preferred style for the interest rate constant would be

```
final static double INTEREST_RATE = 0.05;
```

This is the style that is generally used in Java's standard classes, which define many named constants. For example, the `Math` class defines a named constant `PI` to represent the mathematical constant of that name. Since it is a member of the `Math` class, you would have to refer to it as `Math.PI` in your own programs. Many constants are provided to give meaningful names to be used as parameters in subroutine calls. For example, a standard class named `Font` contains named constants `Font.PLAIN`, `Font.BOLD`, and `Font.ITALIC`. These constants are used for specifying different styles of text when calling various subroutines in the `Font` class.

Curiously enough, one of the major reasons to use named constants is that it's easy to change the value of a named constant. Of course, the value can't change while the program is running. But between runs of the program, it's easy to change the value in the source code and recompile the program. Consider the interest rate example. It's quite possible that the value of the interest rate is used many times throughout the program. Suppose that the bank changes the interest rate and the program has to be modified. If the literal number 0.05 were used throughout the

program, the programmer would have to track down each place where the interest rate is used in the program and change the rate to the new value. (This is made even harder by the fact that the number 0.05 might occur in the program with other meanings besides the interest rate, as well as by the fact that someone might have used 0.025 to represent half the interest rate.) On the other hand, if the named constant `INTEREST_RATE` is declared and used consistently throughout the program, then only the single line where the constant is initialized needs to be changed.

As an extended example, I will give a new version of the `RandomMosaicWalk` program from the [previous section](#). This version uses named constants to represent the number of rows in the mosaic, the number of columns, and the size of each little square. The three constants are declared as `final static` member variables with the lines:

```
final static int ROWS = 30;           // Number of rows in mosaic.
final static int COLUMNS = 30;       // Number of columns in mosaic.
final static int SQUARE_SIZE = 15;    // Size of each square in mosaic.
```

The rest of the program is carefully modified to use the named constants. For example, in the new version of the program, the Mosaic window is opened with the statement

```
Mosaic.open(ROWS, COLUMNS, SQUARE_SIZE, SQUARE_SIZE);
```

Sometimes, it's not easy to find all the places where a named constants needs to be used. It's always a good idea to run a program using several different values for any named constants, 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.

```
public class RandomMosaicWalk2 {

    /*
     * This program shows 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.
     */

    final static int ROWS = 30;           // Number of rows in mosaic.
    final static int COLUMNS = 30;       // Number of columns in mosaic.
    final static int SQUARE_SIZE = 15;    // Size of each square in mosaic.

    static int currentRow; // row currently containing the disturbance
    static int currentColumn; // column currently containing disturbance

    public static void main(String[] args) {
        // Main program creates the window, fills it with
        // random colors, then moves the disturbance in
        // a random walk around the window.
        Mosaic.open(ROWS, COLUMNS, SQUARE_SIZE, SQUARE_SIZE);
        fillWithRandomColors();
        currentRow = ROWS / 2; // start at center of window
        currentColumn = COLUMNS / 2;
        while (Mosaic.isOpen()) {
            changeToRandomColor(currentRow, currentColumn);
            randomMove();
            Mosaic.delay(20);
        }
    } // end of main()

    static void fillWithRandomColors() {
        // Precondition: The mosaic window is open.
```

```

        // Postcondition: Each rectangle has been set to a
        //                    random color
        for (int row=0; row < ROWS; row++) {
            for (int column=0; column < COLUMNS; column++) {
                changeToRandomColor(row, column);
            }
        }
    } // end of fillWithRandomColors()

    static void changeToRandomColor(int rowNum, int colNum) {
        // Precondition:  rowNum and colNum are in the valid range
        //                    of row and column numbers.
        // Postcondition: The rectangle in the specified row and
        //                    column has been changed to a random color.
        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);
    } // end of changeToRandomColor()

    static void randomMove() {
        // Precondition:  The global variables currentRow and currentColumn
        //                    specify a valid position in the grid.
        // Postcondition: currentRow or currentColumn is changed to
        //                    one of the neighboring positions in the grid,
        //                    up, down, left, or right from the previous
        //                    position. (If this moves the position outside
        //                    the grid, then it is moved to the opposite edge
        //                    of the window.)
        int directionNum; // Randomly set to 0, 1, 2, or 3
                           //                    to choose direction.
        directionNum = (int)(4*Math.random());
        switch (directionNum) {
            case 0: // move up
                currentRow--;
                if (currentRow < 0)
                    currentRow = ROWS - 1;
                break;
            case 1: // move right
                currentColumn++;
                if (currentColumn >= COLUMNS)
                    currentColumn = 0;
                break;
            case 2: // move down
                currentRow ++;
                if (currentRow >= ROWS)
                    currentRow = 0;
                break;
            case 3: // move left
                currentColumn--;
                if (currentColumn < 0)
                    currentColumn = COLUMNS - 1;
                break;
        }
    } // end of randomMove()
} // end of class RandomMosaicWalk2

```

Naming and Scope Rules

When a variable declaration is executed, memory is allocated for that variable. The variable name can be used in at least some part of the program source code to refer to that memory or to the data that is stored in the memory. The portion of the program source code where the variable name is valid is called the **scope** of the variable. Similarly, we can refer to the scope of subroutine names and formal parameter names.

For static member subroutines, scope is straightforward. The scope of a static subroutine is the entire source code of the class in which it is defined. That is, it is possible to call the subroutine from any point in the class. 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 later.

For a variable that is declared as a static member variable in a class, the situation is similar, but with one complication. It is legal to have a local variable or a formal parameter that has the same name as a member variable. In that case, within the scope of the local variable or parameter, the member variable is **hidden**. Consider, for example, a class named `Game` that has the form:

```
public class Game {

    static int count; // member variable

    static void playGame() {
        int count; // local variable
        .
        .    // Some statements to define playGame()
        .
    }

    .
    .    // More variables and subroutines.
    .

} // end Game
```

In the statements that make up the body of the `playGame()` subroutine, the name "count" refers to the local variable. In the rest of the `Game` class, "count" refers to the member variable, unless hidden by other local variables or parameters named `count`. However, there is one further complication. The member variable named `count` can also be referred to by the full name `Game.count`. Usually, the full name is only used outside the class where `count` is defined. However, there is no rule against using it inside the class. The full name, `Game.count`, can be used inside the `playGame()` subroutine to refer to the member variable. So, the full scope rule for static member variables is that the scope of a 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 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 a formal parameter name. However, please remember that not everything that is possible is a good idea!

End of Chapter 4

[[Next Chapter](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Programming Exercises For Chapter 4

THIS PAGE CONTAINS programming exercises based on material from [Chapter 4](#) of this [on-line Java textbook](#). Each exercise has a link to a discussion of one possible solution of that exercise.

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 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. You can assume that the parameter is one of the possible totals: 2, 3, ..., 12. 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? 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 5: The sample program [RandomMosaicWalk.java](#) from [Section 4.6](#) shows a "disturbance" that wanders around a grid of colored squares. When the disturbance visits a square, the color of that square is changed. The applet at the bottom of [Section 4.7](#) shows a variation on this idea. In this applet, all the squares start out with the default color, black. Every time the disturbance visits a square, a small amount is added to the red component of the color of that square. Write a subroutine that will add 25 to the red component of one of the squares in the mosaic. The row and column numbers of the square should be passed as parameters to the subroutine. Recall that you can discover the current red component of the square in row *r* and column *c* with the function call `Mosaic.getRed(r,c)`. Use your subroutine as a substitute for the `changeToRandomColor()` subroutine in the program [RandomMosaicWalk2.java](#). (This is the improved version of the program from Section 4.7 that uses named constants for the number of rows, number of columns, and square size.) Set the number of rows and the number of columns to 80. Set the square size to 5.

[See the solution!](#)

Exercise 6: For this exercise, you will write a program that has the same behavior as the following applet. Your program will be based on the non-standard `Mosaic` class, which was described in [Section 4.6](#). (Unfortunately, the applet doesn't look too good on many versions of Java.)

The applet shows a rectangle that grows from the center of the applet to the edges, getting brighter as it grows. The rectangle is made up of the little squares of the mosaic. You should first write a subroutine that draws a rectangle on a `Mosaic` window. More specifically, write a subroutine named `rectangle` such that the subroutine call statement

```
rectangle(top,left,height,width,r,g,b);
```

will call `Mosaic.setColor(row,col,r,g,b)` for each little square that lies on the outline of a rectangle. The topmost row of the rectangle is specified by `top`. The number of rows in the rectangle is specified by `height` (so the bottommost row is `top+height-1`). The leftmost column of the rectangle is specified by `left`. The number of columns in the rectangle is specified by `width` (so the rightmost column is `left+width-1`).

The animation loops through the same sequence of steps over and over. In one step, a rectangle is drawn in gray (that is, with all three color components having the same value). There is a pause of 200 milliseconds

so the user can see the rectangle. Then the very same rectangle is drawn in black, effectively erasing the gray rectangle. Finally, the variables giving the top row, left column, size, and color level of the rectangle are adjusted to get ready for the next step. In the applet, the color level starts at 50 and increases by 10 after each step. You might want to make a subroutine that does one loop through all the steps of the animation.

The `main()` routine simply opens a Mosaic window and then does the animation loop over and over until the user closes the window. There is a 1000 millisecond delay between one animation loop and the next. Use a Mosaic window that has 41 rows and 41 columns. (I advise you not to use named constants for the numbers of rows and columns, since the problem is complicated enough already.)

[See the solution!](#)

[[Chapter Index](#) | [Main Index](#)]

Quiz Questions For Chapter 4

THIS PAGE CONTAINS A SAMPLE quiz on material from [Chapter 4](#) of this [on-line Java textbook](#). You should be able to answer these questions after studying that chapter. Sample answers to all the quiz questions can be found [here](#).

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 a useful tool 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: 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 8: Write a `main()` routine that uses the subroutine that you wrote for Question 7 to output 10 lines of stars with 1 star in the first line, 2 stars in the second line, and so on, as shown below.

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

Question 9: Write a function named `countChars` that has a `String` and a `char` as parameters. The function should count the number of times the character occurs in the string, and it should return the result as the value of the function.

Question 10: Write a subroutine with three parameters of type `int`. The subroutine should determine which of its parameters is smallest. The value of the smallest parameter should be returned as the value of the subroutine.

Chapter 5

Programming in the Large II Objects and Classes

WHEREAS A SUBROUTINE represents a single task, an object can encapsulate both data (in the form of instance variables) and a number of different tasks or "behaviors" related to that data (in the form of instance methods). Therefore objects provide another, more sophisticated type of structure that can be used to help manage the complexity of large programs.

This chapter covers the creation and use of objects in Java. Section 4 covers the central ideas of object-oriented programming: inheritance and polymorphism. However, in this textbook, we will generally use these ideas in a limited form, by creating independent classes and building on existing classes rather than by designing entire hierarchies of classes from scratch. Sections 5 and 6 cover some of the many details of object oriented programming in Java. Although these details are used occasionally later in the book, you might want to skim through them now and return to them later when they are actually needed.

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: [Inheritance, Polymorphism, and Abstract Classes](#)
- Section 5: [this and super](#)
- Section 6: [Interfaces, Nested Classes and Other Details](#)
- [Programming Exercises](#)
- [Quiz on this Chapter](#)

[[First Section](#) | [Next Chapter](#) | [Previous Chapter](#) | [Main Index](#)]

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.

Objects are closely related to classes. We have already been working with classes for several chapters, and we have seen that a class can contain variables and subroutines. If an object is also a collection of variables and subroutines, how do they differ from classes? And why does it require a different type of thinking to understand and use them effectively? In the one section where we worked with objects rather than classes, [Section 3.7](#), 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 for constructing objects. The non-static parts of the class specify, or describe, what variables and subroutines the objects will contain. This is part of the explanation of how objects differ from classes: Objects are created and destroyed as the program runs, and there can be many objects with the same structure, if they are created using the same class.

Consider a simple class whose job is to group together a few static member variables. For example, the following class could be used to store information about the person who is using the program:

```
class UserData {
    static String name;
    static int age;
}
```

In a program that uses this class, there is only one copy of each of the variables `UserData.name` and `UserData.age`. There can only be one "user," since we only have memory space to store data about one user. The class, `UserData`, and the variables it contains exist as long as the program runs. Now, consider a similar class that includes non-static variables:

```
class PlayerData {
    String name;
```

```

        int age;
    }

```

In this case, there is no such variable as `PlayerData.name` or `PlayerData.age`, since `name` and `age` are not static members of `PlayerData`. So, there is nothing much in the class at all -- except the potential to create objects. But, it's a lot of potential, since it can be used to create any number of objects! Each object will have its **own variables** called `name` and `age`. There can be many "players" because we can make new objects to represent new players on demand. A program might use this class to store information about multiple players in a game. Each player has a name and an age. When a player joins the game, a new `PlayerData` object can be created to represent that player. If a player leaves the game, the `PlayerData` object that represents that player can be destroyed. A system of objects in the program is being used to dynamically model what is happening in the game. You can't do this with "static" variables!

In [Section 3.7](#), we worked with applets, which are objects. The reason they didn't seem to be any different from classes is because we were only working with one applet in each class that we looked at. But one class can be used to make many applets. Think of an applet that scrolls a message across a Web page. There could be several such applets on the same page, all created from the same class. If the scrolling message in the applet is stored in a non-static variable, then each applet will have its own variable, and each applet can show a different message. The situation is even clearer if you think about windows, which, like applets, are objects. As a program runs, many windows might be opened and closed, but all those windows can belong to the same class. Here again, we have a dynamic situation where multiple objects are created and destroyed as a program runs.

An object that belongs to a class is said to be an **instance** of that class. The variables that the object contains are called **instance variables**. The subroutines that the object contains are called **instance methods**. (Recall that in the context of object-oriented programming, "method" is a synonym for "subroutine". From now on, for subroutines in objects, I will prefer the term "method.") For example, if the `PlayerData` class, as defined above, is used to create an object, then that object is an instance of the `PlayerData` class, and `name` and `age` are instance variables in the object. It is important to remember that the class of an object determines the types of the instance variables; however, the actual data is contained inside the individual objects, not the class. Thus, each object has its own set of data.

An applet that scrolls a message across a Web page might include a subroutine named `scroll()`. Since the applet is an object, this subroutine is an instance method of the applet. The source code for the method is in the class that is used to create the applet. Still, it's better to think of the instance method as belonging to the object, not to the class. The non-static subroutines in the class merely specify the instance methods that every object created from the class will contain. The `scroll()` methods in two different applets do the same thing in the sense that they both scroll messages across the screen. But there is a real difference between the two `scroll()` methods. The messages that they scroll can be different. (You might say that the subroutine definition in the class specifies what type of behavior the objects will have, but the specific behavior can vary from object to object, depending on the values of their instance variables.)

As you can see, the static and the non-static portions of a class are very different things and serve very different purposes. Many classes contain only static members, or only non-static. However, it is possible to mix static and non-static members in a single class, and we'll see a few examples later in this chapter where it is reasonable to do so. 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. This terminology is most useful when the class contains both static and non-static members.

So far, I've been talking mostly in generalities, and I haven't given you much idea 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:

```

class Student {

    String name;    // Student's name.
    double test1, test2, test3;    // Grades on three tests.

    double getAverage() {    // compute average test grade
        return (test1 + test2 + test3) / 3;
    }

}    // end of class Student

```

None of the members of this class are declared to be `static`, so the class exists only for creating objects. This class definition says that any object that is an instance of the `Student` class will include instance variables named `name`, `test1`, `test2`, and `test3`, and it will include an instance method named `getAverage()`. The names and tests in different objects will generally have different values. When called for a particular student, the method `getAverage()` will compute an average using that student's test grades. Different students can have different averages. (Again, this is what it means to say that an instance method belongs to an individual object, not to the class.)

In Java, a class is a type, similar to the built-in types such as `int` and `boolean`. So, a class name can be used to specify the type of a variable in a declaration statement, the type of a formal parameter, or the return type of a function. For example, a program could define a variable named `std` of type `Student` with the statement

```
Student std;
```

However, declaring a variable does not create an object! This is an important point, which is related to this Very Important Fact:

**In Java, no variable can ever hold an object.
A variable can only hold a reference to an object.**

You should think of objects as floating around independently in the computer's memory. In fact, there is a special portion of memory called the **heap** where objects live. Instead of holding an object itself, a variable holds the information necessary to find the object in memory. This information is called a **reference** or **pointer** to the object. In effect, a reference to an object is the address of the memory location where the object is stored. When you use a variable of class type, the computer uses the reference in the variable to find the actual object.

Objects are actually created by an operator called `new`, which creates an object and returns a reference to that object. For example, assuming that `std` is a variable of type `Student`, declared as above, the assignment statement

```
std = new Student();
```

would create a new object which is an instance of the class `Student`, and it would store a reference to that object in the variable `std`. The value of the variable is a reference to the object, not the object itself. It is not quite true, then, to say that the object is the "value of the variable `std`" (though sometimes it is hard to avoid using this terminology). It is certainly not at all true to say that the object is "stored in the variable `std`." The proper terminology is that "the variable `std` **refers to** the object," and I will try to stick to that terminology as much as possible.

So, suppose that the variable `std` refers to an object belonging to the class `Student`. That object has instance variables `name`, `test1`, `test2`, and `test3`. These instance variables can be referred to as `std.name`, `std.test1`, `std.test2`, and `std.test3`. This follows the usual naming convention that when `B` is part of `A`, then the full name of `B` is `A.B`. For example, a program might include the lines

```

System.out.println("Hello, " + std.name
                  + ". Your test grades are:");
System.out.println(std.test1);
System.out.println(std.test2);
System.out.println(std.test3);

```

This would output the name and test grades from the object to which `std` refers. Similarly, `std` can be used to call the `getAverage()` instance method in the object by saying `std.getAverage()`. To print out the student's average, you could say:

```
System.out.println( "Your average is " + std.getAverage() );
```

More generally, you could use `std.name` any place where a variable of type `String` is legal. You can use it in expressions. You can assign a value to it. You can even use it to call subroutines from the `String` class. For example, `std.name.length()` is the number of characters in the student's name.

It is possible for a variable like `std`, whose type is given by a class, to refer to no object at all. We say in this case that `std` holds a **null reference**. The null reference is written in Java as `"null"`. You can store a null reference in the variable `std` by saying

```
std = null;
```

and you could test whether the value of `std` is null by testing

```
if (std == null) . . .
```

If the value of a variable is `null`, then it is, of course, illegal to refer to instance variables or instance methods through that variable -- since there is no object, and hence no instance variables to refer to. For example, if the value of the variable `std` is `null`, then it would be illegal to refer to `std.test1`. If your program attempts to use a null reference illegally like this, the result is an error called a **null pointer exception**.

Let's look at a sequence of statements that work with objects:

```

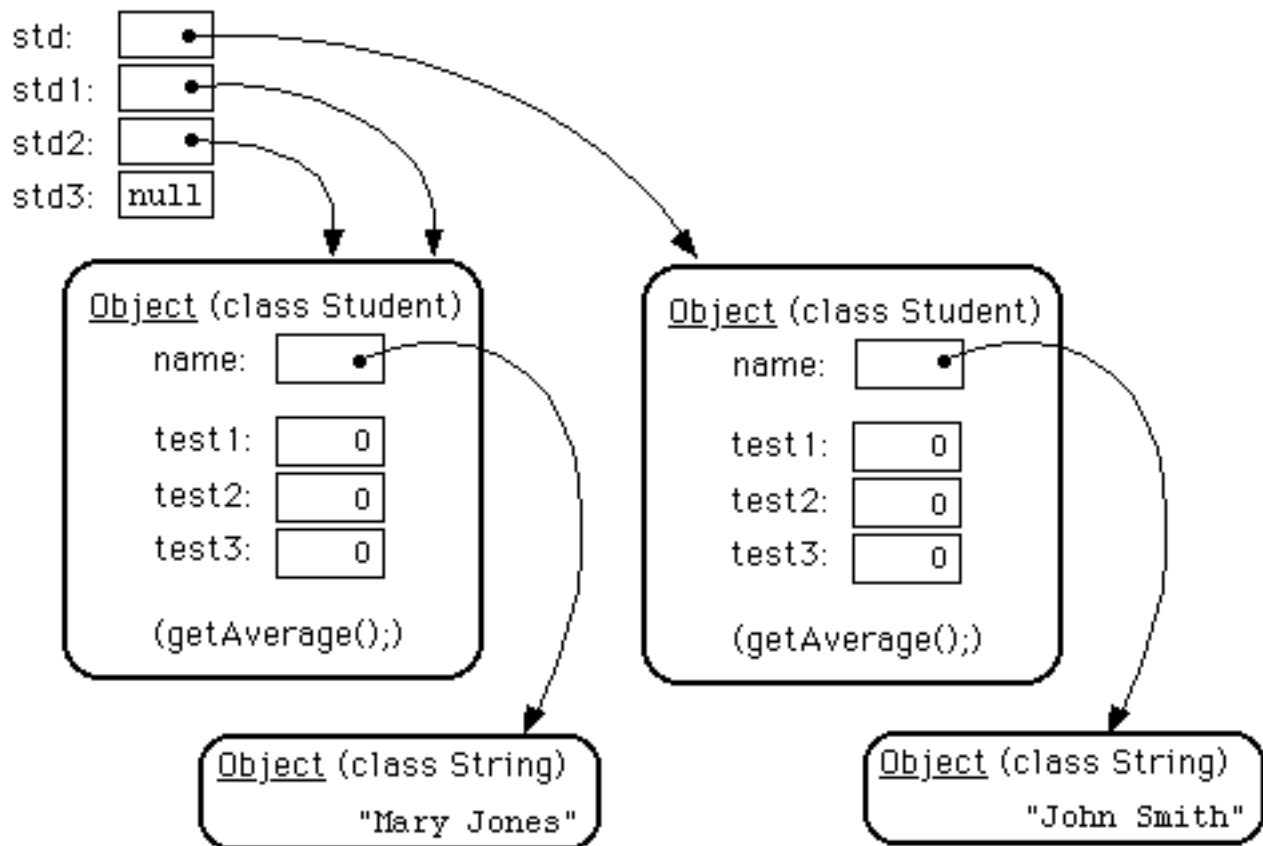
Student std, std1,      // Declare four variables of
    std2, std3;         //   type Student.
std = new Student();     // Create a new object belonging
                        //   to the class Student, and
                        //   store a reference to that
                        //   object in the variable std.
std1 = new Student();    // Create a second Student object
                        //   and store a reference to
                        //   it in the variable std1.
std2 = std1;             // Copy the reference value in std1
                        //   into the variable std2.
std3 = null;             // Store a null reference in the
                        //   variable std3.

std.name = "John Smith"; // Set values of some instance variables.
std1.name = "Mary Jones";

// (Other instance variables have default
//   initial values of zero.)

```

After the computer executes these statements, the situation in the computer's memory looks like this:



This picture shows variables as little boxes, labeled with the names of the variables. Objects are shown as boxes with round corners. When a variable contains a reference to an object, the value of that variable is shown as an arrow pointing to the object. The variable `std3`, with a value of `null`, doesn't point anywhere. The arrows from `std1` and `std2` both point to the same object. This illustrates a Very Important Point:

**When one object variable is assigned
to another, only a reference is copied.
The object referred to is not copied.**

When the assignment `"std2 = std1;"` was executed, no new object was created. Instead, `std2` was set to refer to the very same object that `std1` refers to. This has some consequences that might be surprising. For example, `std1.name` and `std2.name` refer to exactly 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 be true that the value of `std2.name` is "Mary Jones". There is a potential for a lot of confusion here, but you can help protect yourself from it if you keep telling yourself, "The object is not in the variable. The variable just holds a pointer to the object."

You can test objects for equality and inequality using the operators `==` and `!=`, but here again, the semantics are different from what you are used to. When you make a test `"if (std1 == std2)"`, you are testing whether the values stored in `std1` and `std2` are the same. But the values are references to objects, not objects. So, you are testing whether `std1` and `std2` refer to the same object, that is, whether they point to the same location in memory. This is fine, if its what you want to do. But sometimes, what you want to check is whether the instance variables in the objects have the same values. To do that, you would need to ask whether `"std1.test1 == std2.test1 && std1.test2 == std2.test2 && std1.test3 == std2.test3 && std1.name.equals(std2.name)"`

I've remarked previously that `Strings` are objects, and I've shown the strings "Mary Jones" and "John Smith" as objects in the above illustration. A variable of type `String` can only hold a reference

to a string, not the string itself. It could also hold the value `null`, meaning that it does not refer to any string at all. 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 the string it refers to is `"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. 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.

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.
```

Next, suppose that `obj` is a variable that refers to an object. Let's consider what happens when `obj` is passed as an actual parameter to a subroutine. The value of `obj` is assigned to a formal parameter in the subroutine, and the subroutine is executed. The subroutine has no power to change the value stored in the variable, `obj`. It only has a copy of that value. However, that value is a reference to an object. Since the subroutine has a reference to the object, it can change the data stored in the object. After the subroutine ends, `obj` still points to the same object, but the data stored in the object might have changed. Suppose `x` is a variable of type `int` and `stu` is a variable of type `Student`. Compare:

```
void dontChange(int z) {
    z = 42;
}
```

The lines:

```
x = 17;
dontChange(x);
System.out.println(x);
```

output the value 17.

The value of `x` is not changed by the subroutine, which is equivalent to

```
z = x;
z = 42;
```

```
void change(Student s) {
    s.name = "Fred";
}
```

The lines:

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

output the value "Fred".

The value of `stu` is not changed, but `stu.name` is. This is equivalent to

```
s = stu;
s.name = "Fred";
```

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.

An instance variable can be assigned an initial value in its declaration, just like any other variable. For example, consider a class named `PairOfDice`. An object of this class will represent a pair of dice. It will contain two instance variables to represent the numbers showing on the dice and an instance method for rolling the dice:

```
public class PairOfDice {

    public int die1 = 3;    // Number showing on the first die.
    public int die2 = 4;    // Number showing on the second die.

    public void roll() {
        // Roll the dice by setting each of the dice to be
        // a random number between 1 and 6.
        die1 = (int)(Math.random()*6) + 1;
        die2 = (int)(Math.random()*6) + 1;
    }

} // end class PairOfDice
```

The instance variables `die1` and `die2` are initialized to the values 3 and 4 respectively. These initializations are executed whenever a `PairOfDice` object is constructed. It's important to understand when and how this happens. There can be many `PairOfDice` objects. Each time one is created, it gets its own instance variables, and the assignments "`die1 = 3`" and "`die2 = 4`" are executed to fill in the values of those variables. To make this clearer, consider a variation of the `PairOfDice` class:

```
public class PairOfDice {

    public int die1 = (int)(Math.random()*6) + 1;
    public int die2 = (int)(Math.random()*6) + 1;

    public void roll() {
        die1 = (int)(Math.random()*6) + 1;
        die2 = (int)(Math.random()*6) + 1;
    }

} // end class PairOfDice
```

Here, the dice are initialized to random values, as if a new pair of dice were being thrown onto the gaming table. Since the initialization is executed for each new object, a set of random initial values will be computed for each new pair of dice. Different pairs of dice can have different initial values. For initialization of **static member variables**, of course, the situation is quite different. There is only one copy of a static variable, and initialization of that variable is executed just once, when the class is first loaded.

If you don't provide any initial value for an instance variable, a default initial value is provided automatically. Instance variables of numerical type (`int`, `double`, etc.) are automatically initialized to zero if you provide no other values; boolean variables are initialized to `false`; and `char` variables, to the Unicode character with code number zero. An instance variable can also be a variable of object type. For such variables, the default initial value is `null`. (In particular, since `Strings` are objects, the default initial value for `String` variables is `null`.)

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`. 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 a constructor. If the programmer doesn't provide one, then the system will provide a **default constructor**. 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. The only modifiers that can be used on a constructor definition are the access modifiers `public`, `private`, and `protected`. (In particular, a constructor can't be declared `static`.)

However, a constructor does have a subroutine body of the usual form, a block of statements. There are no restrictions on what statements can be used. And it can have a list of formal parameters. In fact, the ability to include parameters is one of the main reasons for using constructors. The parameters can provide data to be used in the construction of the object. For example, a constructor for the `PairOfDice` class could provide the values that are initially showing on the dice. Here is what the class would look like in that case:

```
public class PairOfDice {

    public int die1;    // Number showing on the first die.
    public int die2;    // Number showing on the second die.

    public PairOfDice(int val1, int val2) {
        // Constructor.  Creates a pair of dice that
        // are initially showing the values val1 and val2.
        die1 = val1;    // Assign specified values
        die2 = val2;    // to the instance variables.
    }

    public void roll() {
        // Roll the dice by setting each of the dice to be
        // a random number between 1 and 6.
        die1 = (int)(Math.random()*6) + 1;
        die2 = (int)(Math.random()*6) + 1;
    }

} // end class PairOfDice
```

The constructor is declared as "public PairOfDice(int val1, int val2)...", with no return type and with the same name as the name of the class. This is how the Java compiler recognizes a constructor. The constructor has two parameters, and values for these parameters must be provided when the constructor is called. For example, the expression "new PairOfDice(3,4)" would create a PairOfDice object in which the values of the instance variables die1 and die2 are initially 3 and 4. Of course, in a program, the value returned by the constructor should be used in some way, as in

```
PairOfDice dice;           // Declare a variable of type PairOfDice.
dice = new PairOfDice(1,1); // Let dice refer to a new PairOfDice
                           // object that initially shows 1, 1.
```

Now that we've added a constructor to the PairOfDice class, we can no longer create an object by saying "new PairOfDice()". The system provides a default constructor for a class only if the class definition does not already include a constructor. 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:

```

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

```

This applet simulates this program:

**(Applet "RollTwoPairsConsole" would be displayed here
if Java were available.)**

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!

Unlike other subroutines, a constructor can only be called using the `new` operator, in an expression that has the form

new **class-name** (**parameter-list**)

where the **parameter-list** is possibly empty. I call this an expression because it computes and returns a value, namely a reference to the object that is constructed. Most often, you will store the returned reference in a variable, but it is also legal to use a constructor call in other ways, for example as a parameter in a subroutine call or as part of a more complex expression. Of course, if you don't save the reference in a variable, you won't have any way of referring to the object that was just created.

A constructor call is more complicated than an ordinary subroutine or function call. It is helpful to understand the exact steps that the computer goes through to execute a constructor call:

1. First, the computer gets a block of unused memory in the heap, large enough to hold an object of the specified type.
2. It initializes the instance variables of the object. If the declaration of an instance variable specifies an initial value, then that value is computed and stored in the instance variable. Otherwise, the default initial value is used.
3. The actual parameters in the constructor, if any, are evaluated, and the values are assigned to the formal parameters of the constructor.
4. The statements in the body of the constructor, if any, are executed.
5. A reference to the object is returned as the value of the constructor call.

The end result of this is that you have a reference to a newly constructed object. You can use this reference to get at the instance variables in that object or to call its instance methods.

For another example, let's rewrite the `Student` class that was used in [Section 1](#). I'll add a constructor, and I'll also take the opportunity to make the instance variable, `name`, private.

```
public class Student {

    private String name;           // Student's name.
    public double test1, test2, test3; // Grades on three tests.

    Student(String theName) {
        // Constructor for Student objects;
        // provides a name for the Student.
        name = theName;
    }

    public String getName() {
        // Accessor method for reading value of private
        // instance variable, name.
        return name;
    }

    public double getAverage() {
        // Compute average test grade.
        return (test1 + test2 + test3) / 3;
    }

} // end of class Student
```

An object of type `Student` contains information about some particular student. The constructor in this class has a parameter of type `String`, which specifies the name of that student. Objects of type `Student` can be created with statements such as:

```
std = new Student("John Smith");
```



```
std1 = new Student("Mary Jones");
```

In the original version of this class, the value of `name` had to be assigned by a program after it created the object of type `Student`. There was no guarantee that the programmer would always remember to set the `name` properly. In the new version of the class, there is no way to create a `Student` object except by calling the constructor, and that constructor automatically sets the `name`. The programmer's life is made easier, and whole hordes of frustrating bugs are squashed before they even have a chance to be born.

Another type of guarantee is provided by the `private` modifier. Since the instance variable, `name`, is `private`, there is no way for any part of the program outside the `Student` class to get at the `name` directly. The program sets the value of `name`, indirectly, when it calls the constructor. I've provided a function, `getName()`, that can be used from outside the class to find out the `name` of the student. But I haven't provided any way to change the name. Once a student object is created, it keeps the same name as long as it exists.

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):

```
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 stored in any variable. So there is no way for the program ever to use the object again. It might as well not exist. In fact, the memory occupied by the object should be reclaimed to be used for another purpose.

Java uses a procedure called **garbage collection** to reclaim memory occupied by objects that are no longer accessible to a program. It is the responsibility of the system, not the programmer, to keep track of which objects are "garbage". In the above example, it was very easy to see that the `Student` object had become garbage. Usually, it's much harder. If an object has been used for a while, there might be several references to the object stored in several variables. The object doesn't become garbage until all those references have been dropped.

In many other programming languages, it's the programmer's responsibility to delete the garbage. Unfortunately, keeping track of memory usage is very error-prone, and many serious program bugs are caused by such errors. A programmer might accidentally 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.

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 the full range of built-in classes -- something that takes a lot of time and experience to develop. In the [next chapter](#), we will begin the study of Java's GUI classes, and you will encounter other built-in classes throughout the remainder of this book. But let's take a moment to look at a few built-in classes that you might find useful.

A string can be built up from smaller pieces using the + operator, but this is not very 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 `java.lang.StringBuffer` makes it possible to be efficient about building up a long string from a number of smaller pieces. Like a `String`, a `StringBuffer` contains a sequence of characters. However, it is possible to add new characters onto the end of a `StringBuffer` without making a copy of the data that it already contains. If `buffer` is a variable of type `StringBuffer` and `x` is a value of any type, then the command `buffer.append(x)` will add `x`, converted into a string representation, onto the end of the data that was already in the buffer. This command actually modifies the buffer, rather than making a copy, and that can be done efficiently. A long string can be built up in a `StringBuffer` using a sequence of `append()` commands. When the string is complete, the function `buffer.toString()` will return a copy of the string in the buffer as an ordinary value of type `String`.

A number of useful classes are collected in the package `java.util`. For example, this package contains classes for working with collections of objects (one of the contexts in which wrapper classes for primitive types are useful). We will study these collection classes in [Chapter 12](#). The class `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, to use the `Date` class in this way, 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 [Section 4.5](#) for a discussion of packages and import.)

Finally, I will mention the class `java.util.Random`. An object belonging to this class is a *source of random 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.)

The main point here, again, is that many problems have already been solved, and the solutions are available in Java's standard classes. If you are faced with a task that looks like it should be fairly common, it might be worth looking through a Java reference to see whether someone has already written a subroutine that you can use.

Generalized Software Components

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. We will discuss subclasses in the [next section](#).

Object-oriented Analysis and Design

A large programming project goes through a number of stages, starting with **specification** of the problem to be solved, followed by **analysis** of the problem and **design** of a program to solve it. Then comes **coding**, in which the program's design is expressed in some actual programming language. This is followed by **testing** and **debugging** of the program. After that comes a long period of **maintenance**, which means fixing any new problems that are found in the program and modifying it to adapt it to changing requirements. Together, these stages form what is called the **software life cycle**. (In the real world, the ideal of consecutive stages is seldom if ever achieved. During the analysis stage, it might turn out that the specifications are incomplete or inconsistent. A problem found during testing requires at least a brief return to the coding stage. If the problem is serious enough, it might even require a new design. Maintenance usually involves redoing some of the work from previous stages....)

Large, complex programming projects are only likely to succeed if a careful, systematic approach is adopted during all stages of the software life cycle. The systematic approach to programming, using accepted principles of good design, is called **software engineering**. The software engineer tries to efficiently construct programs that verifiably meet their specifications and that are easy to modify if necessary. There is a wide range of "methodologies" that can be applied to help in the systematic design of programs. (Most

of these methodologies seem to involve drawing little boxes to represent program components, with labeled arrows to represent relationships among the boxes.)

We have been discussing object orientation in programming languages, which is relevant to the coding stage of program development. But there are also object-oriented methodologies for analysis and design. The question in this stage of the software life cycle is, How can one discover or invent the overall structure of a program? As an example of a rather simple object-oriented approach to analysis and design, consider this advice: Write down a description of the problem. Underline all the nouns in that description. The nouns should be considered as candidates for becoming classes or objects in the program design. Similarly, underline all the verbs. These are candidates for methods. This is your starting point. Further analysis might uncover the need for more classes and methods, and it might reveal that subclassing can be used to take advantage of similarities among classes.

This is perhaps a bit simple-minded, but the idea is clear and the general approach can be effective: Analyze the problem to discover the concepts that are involved, and create classes to represent those concepts. The design should arise from the problem itself, and you should end up with a program whose structure reflects the structure of the problem in a natural way.

Programming Examples

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 behaviour of a pair of dice, that is, the ability to be rolled. This class would be reusable in many different programming projects.

On the other hand, the `Student` class from the previous section is not very reusable. It seems to be crafted to represent students in a particular course where the grade will be based on three tests. If there are more tests or quizzes or papers, it's useless. If there are two people in the class who have the same name, we are in trouble (one reason why numerical student ID's are often used). Admittedly, it's much more difficult to develop a general-purpose student class than a general-purpose pair-of-dice class. But this particular `Student` class is good mostly as an example in a programming textbook.

Let's do another example 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). In a typical card game, each player gets a hand of cards. The deck is shuffled and cards are dealt one at a time from the deck and added to the players' hands. In some games, cards can be removed from a hand, and new cards can be added. The game is won or lost depending on the value (ace, 2, ..., king) and suit (spades, diamonds, clubs, hearts) of the cards that a player receives. If we look for nouns in this description, there are several candidates for objects: game, player, hand, card, deck, value, and suit. Of these, the value and the suit of a card are simple values, and they will 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 us two candidates for instance methods in a `Deck` class. Cards can be added to and removed from hands. This gives two candidates for instance methods in a `Hand` class. Cards are relatively passive things, but we need to be able to determine their suits and values. We will discover more instance methods as we go along.

First, we'll design the deck class in detail. When a deck of cards is first created, it contains 52 cards in some standard order. The `Deck` class will need a constructor to create a new deck. The constructor needs no parameters because any new deck is the same as any other. There will be an instance method called `shuffle()` that will rearrange the 52 cards into a random order. The `dealCard()` instance method will

get the next card from the deck. This will be a function with a return type of `Card`, since the caller needs to know what card is being dealt. It has no parameters. 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. 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`:

```
public Deck()
    // Constructor.  Create an unshuffled deck of cards.

public void shuffle()
    // Put all the used cards back into the deck,
    // and shuffle it into a random order.

public int cardsLeft()
    // 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 Card dealCard()
    // Deals one card from the deck and returns it.
```

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 programming. In fact, writing the class involves a programming technique, arrays, which will not be covered until [Chapter 8](#). Nevertheless, you can look at the source code, [Deck.java](#), if you want. And given the source code, you can use the class in your programs without understanding the implementation.

We can do a similar analysis for the `Hand` class. When a hand object is first created, it has no cards in it. An `addCard()` instance method will add a card to the hand. This method needs a parameter of type `Card` to specify which card is being added. For the `removeCard()` method, a parameter is needed to specify which card to remove. But should we specify the card itself ("Remove the ace of spades"), or should we specify the card by its position in the hand ("Remove the third card in the hand")? Actually, we don't have to decide, since we can allow for both options. We'll have two `removeCard()` instance methods, one with a parameter of type `Card` specifying the card to be removed and one with a parameter of type `int` specifying the position of the card in the hand. (Remember that you can have two methods in a class with the same name, provided they have different 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`:

```
public Hand() {
    // Create a Hand object that is initially empty.

public void clear() {
    // Discard all cards from the hand, making the hand empty.
```



```

public void addCard(Card c) {
    // Add the card c to the hand.  c should be non-null.
    // (If c is null, nothing is added to the hand.)

public void removeCard(Card c) {
    // If the specified card is in the hand, it is removed.

public void removeCard(int position) {
    // If the specified position is a valid position in the
    // hand, then the card in that position is removed.

public int getCardCount() {
    // Return the number of cards in the hand.

public Card getCard(int position) {
    // Get the card from the hand in given position, where
    // positions are numbered starting from 0.  If the
    // specified position is not the position number of
    // a card in the hand, then null is returned.

public void sortBySuit() {
    // 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 sortByValue() {
    // Sorts the cards in the hand so that cards of the same
    // value are grouped together.  Cards with the same value
    // are sorted by suit. Note that aces are considered
    // to have the lowest value, 1.

```

Again, you don't yet know enough to implement this class. But given the source code, [Hand.java](#), you can use the class in your own programming projects.

We have covered enough material to write a `Card` class. The class will have a constructor that specifies the value and suit of the card that is being created. There are four suits, which can be represented by the integers 0, 1, 2, and 3. It would be tough to remember which number represents which suit, so I've defined named constants in the `Card` class to represent the four possibilities. For example, `Card.SPADES` is a constant that represents the suit, spades. (These constants are declared to be `public final static ints`. This is one case in which it makes sense to have static members in a class that otherwise has only instance variables and instance methods.) 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. So, cards can be constructed by 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 instance methods `getSuit()` and `getValue()` so that it will be possible to discover the suit and value from outside the class. The instance variables are initialized in the constructor, and are never changed after that. In fact, I've declared the

instance variables `suit` and `value` to be `final`, since they are never changed after they are initialized. (An instance variable can be declared `final` provided it is either given an initial value in its declaration or is initialized in every constructor in the class.)

Finally, I've added a few convenience methods to the class to make it easier to print out cards in a human-readable form. For example, I want to be able to print out the suit of a card as the word "Diamonds", rather than as the meaningless code number 2, which is used in the class to represent diamonds. Since this is something that I'll probably have to do in many programs, it makes sense to include support for it in the class. So, I've provided instance methods `getSuitAsString()` and `getValueAsString()` to return string representations of the suit and value of a card. Finally, there is an instance method `toString()` that returns a string with both the value and suit, such as "Queen of Hearts". There is a good reason for calling this method `toString()`. When any object is output with `System.out.print()`, the object's `toString()` method is called to produce the string representation of the object. For example, if `card` refers to an object of type `Card`, then `System.out.println(card)` is equivalent to `System.out.println(card.toString())`. Similarly, if an object is appended to a string using the `+` operator, the object's `toString()` method is used. Thus,

```
System.out.println( "Your card is the " + card );
```

is equivalent to

```
System.out.println( "Your card is the " + card.toString() );
```

If the card is the queen of hearts, either of these will print out "Your card is the Queen of Hearts".

Here is the complete `Card` class. It is general enough to be highly reusable, so the work that went into designing, writing, and testing it pays off handsomely in the long run.

```
/*
   An object of class card represents one of the 52 cards in a
   standard deck of playing cards.  Each card has a suit and
   a value.
*/

public class Card {

    public final static int SPADES = 0,      // Codes for the 4 suits.
                          HEARTS = 1,
                          DIAMONDS = 2,
                          CLUBS = 3;

    public final static int ACE = 1,         // Codes for non-numeric cards.
                          JACK = 11,        // Cards 2 through 10 have
                          QUEEN = 12,       // their numerical values
                          KING = 13;       // for their codes.

    private final int suit;  // The suit of this card, one of the
                           // four constants: SPADES, HEARTS,
                           // DIAMONDS, CLUBS.

    private final int value; // The value of this card, from 1 to 13.

    public Card(int theValue, int theSuit) {
        // Construct a card with the specified value and suit.
        // Value must be between 1 and 13. Suit must be between
        // 0 and 3. If the parameters are outside these ranges,
```

```

        // the constructed card object will be invalid.
        value = theValue;
        suit = theSuit;
    }

    public int getSuit() {
        // Return the int that codes for this card's suit.
        return suit;
    }

    public int getValue() {
        // Return the int that codes for this card's value.
        return value;
    }

    public String getSuitAsString() {
        // Return a String representing the card's suit.
        // (If the card's suit is invalid, "??" is returned.)
        switch ( suit ) {
            case SPADES:    return "Spades";
            case HEARTS:    return "Hearts";
            case DIAMONDS:  return "Diamonds";
            case CLUBS:     return "Clubs";
            default:        return "??";
        }
    }

    public String getValueAsString() {
        // Return a String representing the card's value.
        // If the card's value is invalid, "??" is returned.
        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";
            case 13:   return "King";
            default:   return "??";
        }
    }

    public String toString() {
        // Return a String representation of this card, such as
        // "10 of Hearts" or "Queen of Spades".
        return getValueAsString() + " of " + getSuitAsString();
    }

} // end class Card

```

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 subroutine that plays one game of `HighLow`. This subroutine has a return value that represents the user's score in the game. The `main()` routine lets the user play several games of `HighLow`. At the end, it reports the user's average score.

I won't go through the development of the algorithms used in this program, but I encourage you to read it carefully and make sure that you understand how it works. Here is the program:

```
/*
   This program lets the user play HighLow, a simple card game
   that is described in the output statements at the beginning of
   the main() routine. After the user plays several games,
   the user's average score is reported.
*/

public class HighLow {

    public static void main(String[] args) {

        TextIO.putln("This program lets you play the simple card game,");
        TextIO.putln("HighLow. A card is dealt from a deck of cards.");
        TextIO.putln("You have to predict whether the next card will be");
        TextIO.putln("higher or lower. Your score in the game is the");
        TextIO.putln("number of correct predictions you make before");
        TextIO.putln("you guess wrong.");
        TextIO.putln();

        int gamesPlayed = 0;           // Number of games user has played.
        int sumOfScores = 0;           // The sum of all the scores from
                                      // all the games played.
        double averageScore;          // Average score, computed by dividing
                                      // sumOfScores by gamesPlayed.
        boolean playAgain;            // Record user's response when user is
                                      // asked whether he wants to play
                                      // another game.

        do {
            int scoreThisGame;         // Score for one game.
            scoreThisGame = play();     // Play the game and get the score.
            sumOfScores += scoreThisGame;
            gamesPlayed++;
            TextIO.put("Play again? ");
            playAgain = TextIO.getlnBoolean();
        } while (playAgain);
    }
}
```

```

        averageScore = ((double)sumOfScores) / gamesPlayed;

        TextIO.putln();
        TextIO.putln("You played " + gamesPlayed + " games.");
        TextIO.putln("Your average score was " + averageScore);
    } // end main()

    static int play() {
        // Lets the user play one game of HighLow, and returns the
        // user's score in the game.

        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();
        correctGuesses = 0;
        currentCard = deck.dealCard();
        TextIO.putln("The first card is the " + currentCard);

        while (true) { // Loop ends when user's prediction is wrong.

            /* Get the user's prediction, 'H' or 'L'. */

            TextIO.put("Will the next card be higher (H) or lower (L)? ");
            do {
                guess = TextIO.getlnChar();
                guess = Character.toUpperCase(guess);
                if (guess != 'H' && guess != 'L')
                    TextIO.put("Please respond with H or L: ");
            } while (guess != 'H' && guess != 'L');

            /* Get the next card and show it to the user. */

            nextCard = deck.dealCard();
            TextIO.putln("The next card is " + nextCard);

            /* Check the user's prediction. */

```

```

        if (nextCard.getValue() == currentCard.getValue()) {
            TextIO.putln("The value is the same as the previous card.");
            TextIO.putln("You lose on ties.  Sorry!");
            break;  // End the game.
        }
        else if (nextCard.getValue() > currentCard.getValue()) {
            if (guess == 'H') {
                TextIO.putln("Your prediction was correct.");
                correctGuesses++;
            }
            else {
                TextIO.putln("Your prediction was incorrect.");
                break;  // End the game.
            }
        }
        else {  // nextCard is lower
            if (guess == 'L') {
                TextIO.putln("Your prediction was correct.");
                correctGuesses++;
            }
            else {
                TextIO.putln("Your prediction was incorrect.");
                break;  // End the game.
            }
        }
    }

    /* To set up for the next iteration of the loop, the nextCard
       becomes the currentCard, since the currentCard has to be
       the card that the user sees, and the nextCard will be
       set to the next card in the deck after the user makes
       his prediction.  */

    currentCard = nextCard;
    TextIO.putln();
    TextIO.putln("The card is " + currentCard);

} // end of while loop

TextIO.putln();
TextIO.putln("The game is over.");
TextIO.putln("You made " + correctGuesses
              + " correct predictions.");
TextIO.putln();

return correctGuesses;

} // end play()

} // end class HighLow

```

Here is an applet that simulates the program:

(Applet "HighLowConsole" would be displayed here
if Java were available.)

[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Section 5.4

Inheritance, Polymorphism, and Abstract Classes

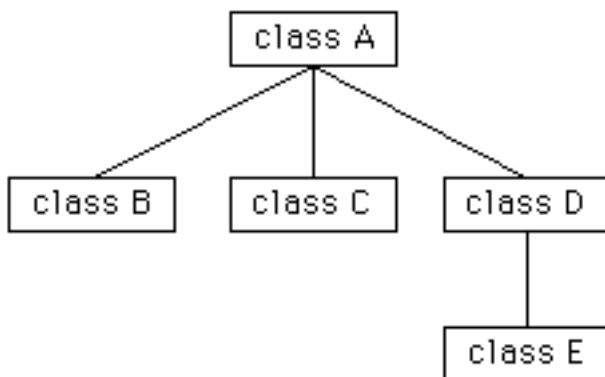
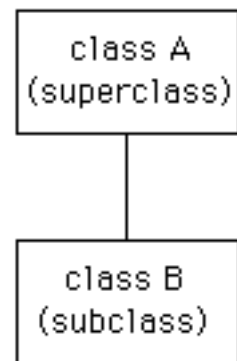
A CLASS REPRESENTS A SET OF OBJECTS which share the same structure and behaviors. The class determines the structure of objects by specifying variables that are contained in each instance of the class, and it determines behavior by providing the instance methods that express the behavior of the objects. 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**.

The topics covered in this section are relatively advanced aspects of object-oriented programming. Any programmer should know what is meant by subclass, inheritance, and polymorphism. However, it will probably be a while before you actually do anything with inheritance except for extending classes that already exist.

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.) 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.

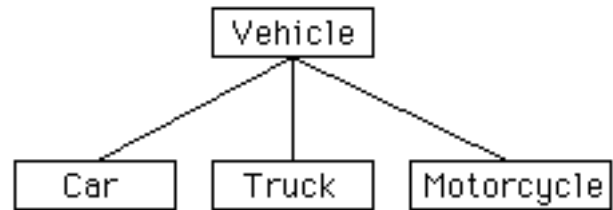
In Java, when you create a new class, you can declare that it is a subclass of an existing class. If you are defining a class named "B" and you want it to be a subclass of a class named "A", you would write

```
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 to the left, 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.

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. The `Vehicle` class could include instance variables such as `registrationNumber` and `owner` and instance methods such as `transferOwnership()`. These are variables and methods common to all vehicles. 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 `numberOfAxels`, 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 Java program would look, in outline, like this:



```

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 numberOfAxels;
    . . .
}
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();
```

(Note that, as with any variable, it is OK to declare a variable and initialize it in a single statement. This is equivalent to the declaration `"Car myCar;"` followed by the assignment statement `"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`. This brings us to the following Important Fact:

**A variable that can hold a reference
to an object of class A can also hold a reference
to an object belonging to any subclass of A.**

The practical effect of this in our example is that an object of type `Car` can be assigned to a variable of type

Vehicle. That is, it would be legal to say

```
Vehicle myVehicle = myCar;
```

or even

```
Vehicle myVehicle = new Car();
```

After either of these statements, the variable `myVehicle` holds a reference to a `Vehicle` object that happens to be an instance of the subclass, `Car`. The object "remembers" that it is in fact a `Car`, and not just a `Vehicle`. Information about the actual class of an object is stored as part of that object. It is even possible to test whether a given object belongs to a given class, using the `instanceof` operator. The test:

```
if (myVehicle instanceof Car) ...
```

determines whether the object referred to by `myVehicle` is in fact a car.

On the other hand, the assignment statement

```
myCar = myVehicle;
```

would be illegal because `myVehicle` could potentially refer to other types of vehicles that are not cars. This is similar to a problem we saw previously in [Section 2.5](#): 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;
```

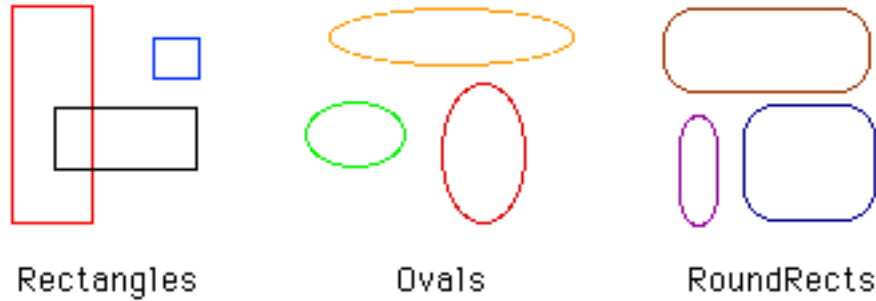
and you could even refer to `((Car)myVehicle).numberOfDoors`. As an example of how this could be used in a program, suppose that you want to print out relevant data about a vehicle. You could say:

```
System.out.println("Vehicle Data:");
System.out.println("Registration number:  "
                  + myVehicle.registrationNumber);
if (myVehicle instanceof Car) {
    System.out.println("Type of vehicle:  Car");
    Car c;
    c = (Car)myVehicle;
    System.out.println("Number of doors:  " + c.numberOfDoors);
}
else if (myVehicle instanceof Truck) {
    System.out.println("Type of vehicle:  Truck");
    Truck t;
    t = (Truck)myVehicle;
    System.out.println("Number of axels:  " + t.numberOfAxels);
}
else if (myVehicle instanceof Motorcycle) {
    System.out.println("Type of vehicle:  Motorcycle");
    Motorcycle m;
    m = (Motorcycle)myVehicle;
    System.out.println("Has a sidecar:    " + m.hasSidecar);
}
```

Note that for object types, when the computer executes a program, it checks whether type-casts are valid. So, for example, if `myVehicle` refers to an object of type `Truck`, then the type cast `(Car)myVehicle`

will produce an error.

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.



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. It could include instance methods for changing the color, position, and size of a shape. Changing the color, for example, might involve changing the value of an instance variable, and then redrawing the shape in its new color:

```
class Shape {
    Color color;    // Color of the shape. (Recall that class Color
                   // is defined in package java.awt. Assume
                   // that this class has been imported.)

    void setColor(Color newColor) {
        // Method to change the color of the shape.
        color = newColor; // change value of instance variable
        redraw(); // redraw shape, which will appear in new color
    }

    void redraw() {
        // method for drawing the shape
        ??? // what commands should go here?
    }

    . . .          // more instance variables and methods
} // end of class Shape
```

Now, you might see a problem here with the method `redraw()`. The problem is that each different type of shape is drawn differently. The method `setColor()` can be called for any type of shape. How does the computer know which shape to draw when it executes the `redraw()`? Informally, we can answer the question like this: The computer executes `redraw()` by asking the shape to redraw itself. Every shape object knows what it has to do to redraw itself.

In practice, this means that each of the specific shape classes has its own `redraw()` method:

```
class Rectangle extends Shape {
    void redraw() {
        . . . // commands for drawing a rectangle
    }
    . . . // possibly, more methods and variables
}
```

```

class Oval extends Shape {
    void redraw() {
        . . . // commands for drawing an oval
    }
    . . . // possibly, more methods and variables
}
class RoundRect extends Shape {
    void redraw() {
        . . . // commands for drawing a rounded rectangle
    }
    . . . // possibly, more methods and variables
}

```

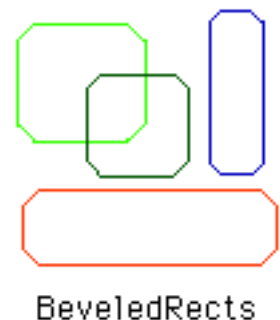
If `oneShape` is a variable of type `Shape`, it could refer to an object of any of the types, `Rectangle`, `Oval`, or `RoundRect`. As a program executes, and the value of `oneShape` changes, it could even refer to objects of different types at different times! Whenever the statement

```
oneShape.redraw();
```

is executed, the `redraw` method that is actually called is the one appropriate for the type of object to which `oneShape` actually refers. There may be no way of telling, from looking at the text of the program, what shape this statement will draw, since it depends on the value that `oneShape` happens to have when the program is executed. Even more is true. Suppose the statement is in a loop and gets executed many times. If the value of `oneShape` changes as the loop is executed, it is possible that the very same statement "`oneShape.redraw();`" will call different methods and draw different shapes as it is executed over and over. We say that the `redraw()` method is **polymorphic**. A method is polymorphic if the action performed by the method depends on the actual type of the object to which the method is applied. Polymorphism is one of the major distinguishing features of object-oriented programming.

Perhaps this becomes more understandable if we change our terminology a bit: In object-oriented programming, calling a method is often referred to as sending a **message** to an object. The object responds to the message by executing the appropriate method. The statement "`oneShape.redraw();`" is a message to the object referred to by `oneShape`. Since that object knows what type of object it is, it knows how it should respond to the message. From this point of view, the computer always executes "`oneShape.redraw();`" in the same way: by sending a message. The response to the message depends, naturally, on who receives it. From this point of view, objects are active entities that send and receive messages, and polymorphism is a natural, even necessary, part of this view. Polymorphism just means that different objects can respond to the same message in different ways.

One of the most beautiful things about polymorphism is that it lets code that you write do things that you didn't even conceive of, at the time you wrote it. If for some reason, I decide that I want to add beveled rectangles to the types of shapes my program can deal with, I can write a new subclass, `BeveledRect`, of class `Shape` and give it its own `redraw()` method. Automatically, code that I wrote previously -- such as the statement `oneShape.redraw()` -- can now suddenly start drawing beveled rectangles, even though the beveled rectangle class didn't exist when I wrote the statement!



In the statement "`oneShape.redraw();`", the `redraw` message is sent to the object `oneShape`. Look back at the method from 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 is the `redraw()` method from the `Rectangle` class that is executed. If the object is an oval, then it is the `redraw()` method from the `Oval` class. This is what you should expect, but it means that the `redraw();` statement in the `setColor()` method does not necessarily call the `redraw()` method in the `Shape` class! The `redraw()` method that is executed could be in any subclass of `Shape`.

Again, this is not a real surprise if you think about it in the right way. Remember that an instance method is always contained in an object. The class only contains the source code for the method. When a `Rectangle` object is created, it contains a `redraw()` method. The source code for that method is in the `Rectangle` class. The object also contains a `setColor()` method. Since the `Rectangle` class does not define a `setColor()` method, the source code for the rectangle's `setColor()` method comes from the superclass, `Shape`. But even though the source codes for the two methods are in different classes, the methods themselves are part of the same object. When the rectangle's `setColor()` method is executed and calls `redraw()`, the `redraw()` method that is executed is the one in the same object.

Whenever a `Rectangle`, `Oval`, or `RoundRect` object has to draw itself, it is the `redraw()` method in the appropriate class that is executed. This leaves open the question, What does the `redraw()` method in the `Shape` class do? How should it be defined?

The answer may be surprising: We should leave it blank! The fact is that the class `Shape` represents the abstract idea of a shape, and there is no way to draw such a thing. Only particular, concrete shapes like rectangles and ovals can be drawn. So, why should there even be a `redraw()` method in the `Shape` class? Well, it has to be there, or it would be illegal to call it in the `setColor()` method of the `Shape` class, and it would be illegal to write "`oneShape.redraw();`", where `oneShape` is a variable of type `Shape`. The compiler would complain that `oneShape` is a variable of type `Shape` and there's no `redraw()` method in the `Shape` class.

Nevertheless the version of `redraw()` in the `Shape` class will never 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.

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 of `Shape`. 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 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:

```
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 done this, it becomes illegal to try to create actual objects of type `Shape`, and the computer will report an error if you try to do so.

In Java, every class that you declare has a superclass. If you don't specify a superclass, then the superclass is automatically taken to be `Object`, a predefined class that is part of the package `java.lang`. (The class `Object` itself has no superclass, but it is the only class that has this property.) Thus,

```
class myClass { . . .
```

is exactly equivalent to

```
class myClass extends Object { . . .
```

Every other class is, directly or indirectly, a subclass of `Object`. This means that any object, belonging to any class whatsoever, can be assigned to a variable of type `Object`. The class `Object` represents very general properties that are shared by all objects, belonging to any class. `Object` is the most abstract class of all!

The `Object` class actually finds a use in some cases where objects of a very general sort are being manipulated. For example, java has a standard class, `java.util.ArrayList`, that represents a list of `Objects`. (The `ArrayList` class is in the package `java.util`. If you want to use this class in a program you write, you would ordinarily use an `import` statement to make it possible to use the short name, `ArrayList`, instead of the full name, `java.util.ArrayList`. `ArrayList` is discussed more fully in [Section 8.3](#).) The `ArrayList` class is very convenient, because an `ArrayList` can hold any number of objects, and it will grow, when necessary, as objects are added to it. Since the items in the list are of type `Object`, the list can actually hold objects of any type.

A program that wants to keep track of various `Shapes` that have been drawn on the screen can store those shapes in an `ArrayList`. Suppose that the `ArrayList` is named `listOfShapes`. A shape, `oneShape`, can be added to the end of the list by calling the instance method `"listOfShapes.add(oneShape);"`. The shape could be removed from the list with `"listOfShapes.remove(oneShape);"`. The number of shapes in the list is given by the function `"listOfShapes.size()"`. And it is possible to retrieve the *i*-th object from the list with the function call `"listOfShapes.get(i)"`. (Items in the list are numbered from 0 to `listOfShapes.size() - 1`.) However, note that this method returns an `Object`, not a `Shape`. (Of course, the people who wrote the `ArrayList` class didn't even know about `Shapes`, so the method they wrote could hardly have a return type of `Shape`!) Since you know that the items in the list are, in fact, `Shapes` and not just `Objects`, you can type-cast the `Object` returned by `listOfShapes.get(i)` to be a value of type `Shape`:

```
oneShape = (Shape)listOfShapes.get(i);
```


Let's say, for example, that you want to redraw all the shapes in the list. You could do this with a simple `for` loop, which is lovely example of object-oriented programming and polymorphism:

```
for (int i = 0; i < listOfShapes.size(); i++) {
    Shape s; // i-th element of the list, considered as a Shape
    s = (Shape)listOfShapes.get(i);
    s.redraw();
}
```

It might be worthwhile to look at an applet that actually uses an abstract `Shape` class and an `ArrayList` to hold a list of shapes:

(Applet "ShapeDraw" would be displayed here
if Java were available.)

If you click one of the buttons along the bottom of this applet, a shape will be added to the screen in the upper left corner of the applet. The color of the shape is given by the "pop-up menu" in the lower right. 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 this applet the only time when the actual class of a shape is used is when that shape is added to the screen. Once the shape has been created, it is manipulated entirely as an abstract shape. The routine that implements dragging, for example, works only with variables of type `Shape`. As the `Shape` is being dragged, the dragging routine just calls the `Shape`'s `draw` method each time the shape has to be drawn, so it doesn't have to know how to draw the shape or even what type of shape it is. The object is responsible for drawing itself. If I wanted to add a new type of shape to the program, I would define a new subclass of `Shape`, add another button to the applet, and program the button to add the correct type of shape to the screen. No other changes in the programming would be necessary.

You might want to look at the source code for this applet, [ShapeDraw.java](#), 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 I described in this section. (For example, the `draw()` method used in the applet has a parameter of type `Graphics`. This parameter is required because of the way Java handles all drawing.) I'll return to this example in later chapters when you know more about applets. However, it would still be worthwhile to look at the definition of the `Shape` class and its subclasses in the source code for the applet. You might also check how an `ArrayList` is used to hold a list of shapes.

Extending Existing Classes

We have been discussing subclasses, but so far we have dealt mostly with the theory. In the remainder of this section, I want to emphasize the practical matter of Java syntax by giving an example.

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

```
class subclass-name extends existing-class-name {
    .
    .    // Changes and additions.
    .
}
```


(Of course, the class can optionally be declared to be `public`.)

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 3](#). 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 {

    public int getBlackjackValue() {
        // Returns the value of this hand for the
        // game of Blackjack.

        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();

        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 `getBlackjackValue()`. For example, if `bHand` is a variable of type `BlackjackHand`, then the following are all legal method calls: `bHand.getCardCount()`, `bHand.removeCard(0)`, and `bHand.getBlackjackValue()`.

Inherited variables and methods from the `Hand` class can also be used in the definition of `BlackjackHand` (except for any that are declared to be `private`). The statement `"cards = getCardCount();"` in the above definition of `getBlackjackValue()` calls the instance method `getCardCount()`, which was defined in the `Hand` class.

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.

[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Section 5.5

this and super

ALTHOUGH THE BASIC IDEAS of object-oriented programming are reasonably simple and clear, they are subtle, and they take time to get used to. And unfortunately, beyond the basic ideas there are a lot of details. This section and the [next](#) cover more of those annoying details. You should not necessarily master everything in these two sections the first time through, but you should read it to be aware of what is possible. For the most part, when I need to use this material later in the text, I will explain it again briefly, or I will refer you back to it. In this section, we'll look at two variables, `this` and `super` that are automatically defined in any instance method.

The Special Variables `this` and `super`

A static member of a class has a simple name, which 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, "`System.out`" is a static member variable with simple name "`out`" in the class "`System`". 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 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. Instance members also have full names, but remember that instance variables and methods are actually contained in objects, not classes. The full name of an instance member has to contain a reference to the object that contains the instance member. To get at an instance variable or method from outside the class definition, you need a variable that refers to the object. Then the full name is of the form **variable-name.simple-name**. But suppose you are writing the definition of an instance method in some class. How can you get a reference to the object that contains that instance method? You might need such a reference, for example, if you want to use the full name of an instance variable, because the simple name of the instance variable is hidden by a local variable or parameter.

Java provides a special, predefined variable named "`this`" that you can use for such purposes. The variable, `this`, is used in the source code of an instance method to refer to the object that contains the method. This intent of the name, `this`, is to refer to "this object," the one right here that this very method is in. If `x` is an instance variable in the same object, then `this.x` can be used as a full name for that variable. If `otherMethod()` is an instance method in the same object, then `this.otherMethod()` could be used to call that method. Whenever the computer executes an instance method, it automatically sets the variable, `this`, to refer to the object that contains the method.

One common use of `this` is in constructors. For example:

```
public class Student {

    private String name;    // Name of the student.

    public Student(String name) {
        // Constructor.  Create a student with specified name.
        this.name = name;
    }

    .
    .    // More variables and methods.
    .
}
```

```
}
```

In the constructor, the instance variable called `name` is hidden by a formal parameter. However, the instance variable can still be referred to by its full name, `this.name`. In the assignment statement, the value of the formal parameter, `name`, is assigned to the instance variable, `this.name`. This is considered to be acceptable style: There is no need to dream up cute new names for formal parameters that are just used to initialize instance variables. You can use the same name for the parameter as for the instance variable.

There are other uses for `this`. Sometimes, when you are writing an instance method, you need to pass the object that contains the method to a subroutine, as an actual parameter. In that case, you can use `this` as the actual parameter. For example, if you wanted to print out a string representation of the object, you could say `System.out.println(this);`. Or you could assign the value of `this` to another variable in an assignment statement. In fact, you can do anything with `this` that you could do with any other variable, except change its value.

Java also defines another special variable, named `"super"`, for use in the definitions of instance methods. The variable `super` is for use in a subclass. Like `this`, `super` refers to the object that contains the method. But it's forgetful. It forgets that the object belongs to the class you are writing, and it remembers only that it belongs to the superclass of that class. The point is that the class can contain additions and modifications to the superclass. `super` doesn't know about any of those additions and modifications; it can only be used to refer to methods and variables in the superclass.

Let's say that the class that you are writing contains an instance method named `doSomething()`. Consider the subroutine call statement `super.doSomething()`. Now, `super` doesn't know anything about the `doSomething()` method in the subclass. It only knows about things in the superclass, so it tries to execute a method named `doSomething()` from the superclass. If there is none -- if the `doSomething()` method was an addition rather than a modification -- you'll get a syntax error.

The reason `super` exists is so you can get access to things in the superclass that are *hidden* by things in the subclass. For example, `super.x` always refers to an instance variable named `x` in the superclass. This can be useful for the following reason: If a class contains an instance variable with the same name as an instance variable in its superclass, then an object of that class will actually contain two variables with the same name: one defined as part of the class itself and one defined as part of the superclass. The variable in the subclass does not replace the variable of the same name in the superclass; it merely hides it. The variable from the superclass can still be accessed, using `super`.

When you write a method in a subclass that has the same signature as a method in its superclass, the method from the superclass is hidden in the same way. We say that the method in the subclass **overrides** the method from the superclass. Again, however, `super` can be used to access the method from the superclass.

The major use of `super` is to override a method with a new method that extends the behavior of the inherited method, instead of replacing that behavior entirely. The new method can use `super` to call the method from the superclass, and then it can add additional code to provide additional behavior. As an example, suppose you have a `PairOfDice` class that includes a `roll()` method. Suppose that you want a subclass, `GraphicalDice`, to represent a pair of dice drawn on the computer screen. The `roll()` method in the `GraphicalDice` class should do everything that the `roll()` method in the `PairOfDice` class does. We can express this with a call to `super.roll()`. But in addition to that, the `roll()` method for a `GraphicalDice` object has to redraw the dice to show the new values. The `GraphicalDice` class might look something like this:

```
public class GraphicalDice extends PairOfDice {

    public void roll() {
        // Roll the dice, and redraw them.
```

```

        super.roll(); // Call the roll method from PairOfDice.
        redraw();    // Call a method to draw the dice.
    }

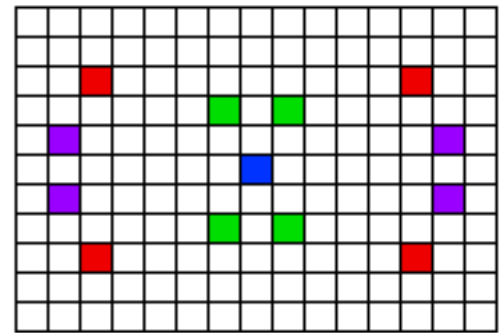
    .
    . // More stuff, including definition of redraw().
    .
}

```

Note that this allows you to extend the behavior of the `roll()` method even if you don't know how the method is implemented in the superclass!

Here is a more complete example. The applet at the end of [Section 4.7](#) shows a disturbance that moves around in a mosaic of little squares. As it moves, the squares it visits become a brighter red. The result looks interesting, but I think it would be prettier if the pattern were symmetric. A symmetric version of the applet is shown at the bottom of the [next section](#). The symmetric applet can be programmed as an easy extension of the original applet.

In the symmetric version, each time a square is brightened, the squares that can be obtained from that one by horizontal and vertical reflection through the center of the mosaic are also brightened. The four red squares in the picture, for example, form a set of such symmetrically placed squares, as do the purple squares and the green squares. (The blue square is at the center of the mosaic, so reflecting it doesn't produce any other squares; it's its own reflection.)



The original applet is defined by the class `RandomBrighten`. This class uses features of Java that you won't learn about for a while yet, but the actual task of brightening a square is done by a single method called `brighten()`. If `row` and `col` are the row and column numbers of a square, then "`brighten(row, col);`" increases the brightness of that square. All we need is a subclass of `RandomBrighten` with a modified `brighten()` routine. Instead of just brightening one square, the modified routine will also brighten the horizontal and vertical reflections of that square. But how will it brighten each of the four individual squares? By calling the `brighten()` method from the original class. It can do this by calling `super.brighten()`.

There is still the problem of computing the row and column numbers of the horizontal and vertical reflections. To do this, you need to know the number of rows and the number of columns. The `RandomBrighten` class has instance variables named `ROWS` and `COLUMNS` to represent these quantities. Using these variables, it's possible to come up with formulas for the reflections, as shown in the definition of the `brighten()` method below.

Here's the complete definition of the new class:

```

public class SymmetricBrighten extends RandomBrighten {

    void brighten(int row, int col) {
        // Brighten the specified square and its horizontal
        // and vertical reflections. This overrides the brighten
        // method from the RandomBrighten class, which just
        // brightens one square.
        super.brighten(row, col);
        super.brighten(ROWS - 1 - row, col);
        super.brighten(row, COLUMNS - 1 - col);
        super.brighten(ROWS - 1 - row, COLUMNS - 1 - col);
    }
}

```

```
} // end class SymmetricBrighten
```

This is the entire source code for the applet!

Constructors in Subclasses

Constructors are not inherited. That is, if you extend an existing class to make a subclass, the constructors in the superclass do **not** become part of the subclass. If you want constructors in the subclass, you have to define new ones from scratch. If you don't define any constructors in the subclass, then the computer will make up a default constructor, with no parameters, for you.

This could be a problem, if there is a constructor in the superclass that does a lot of necessary work. It looks like you might have to repeat all that work in the subclass! This could be a real problem if you don't have the source code to the superclass, and don't know how it works, or if the constructor in the superclass initializes `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...
    .
}
```

This might seem rather technical, but unfortunately it is sometimes necessary. By the way, you can use the special variable `this` in exactly the same way to call another constructor in the same class. This can be useful since it can save you from repeating the same code in several constructors.

[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

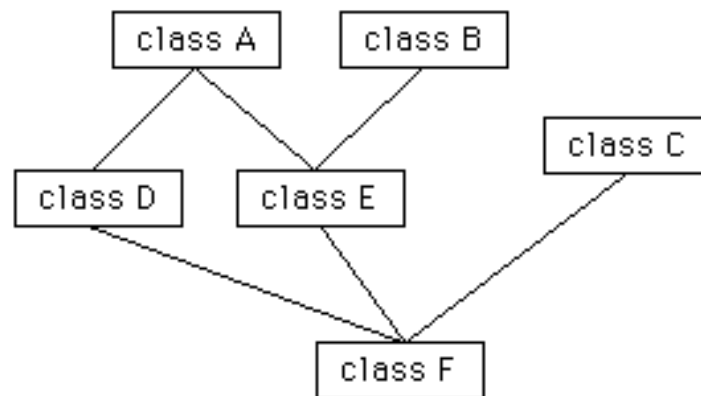
Section 5.6

Interfaces, Nested Classes, and Other Details

THIS SECTION simply pulls together a few more miscellaneous features of object oriented programming in Java. Read it now, or just look through it and refer back to it later when you need this material.

Interfaces

Some object-oriented programming languages, such as C++, allow a class to extend two or more superclasses. This is called **multiple inheritance**. In the illustration below, for example, class E is shown as having both class A and class B as direct superclasses, while class F has three direct superclasses.



Multiple Inheritance (**NOT** allowed in Java)

Such multiple inheritance is **not allowed in Java**. The designers of Java wanted to keep the language reasonably simple, and felt that the benefits of multiple inheritance were not worth the cost in increased complexity. However, Java does have a feature that can be used to accomplish many of the same goals as multiple inheritance: **interfaces**.

We've encountered the term "interface" before, in connection with black boxes in general and subroutines in particular. The interface of a subroutine consists of the name of the subroutine, its return type, and the number and types of its parameters. This is the information you need to know if you want to call the subroutine. A subroutine also has an implementation: the block of code which defines it and which is executed when the subroutine is called.

In Java, `interface` is a reserved word with an additional, technical meaning. An "interface" in this sense consists of a set of subroutine interfaces, without any associated implementations. A class can **implement** an interface by providing an implementation for each of the subroutines specified by the interface. Here is an example of a very simple Java interface:

```
public interface Drawable {
    public void draw(Graphics g);
}
```

This looks much like a class definition, except that the implementation of the method `draw()` is omitted. A class that implements the interface, `Drawable`, must provide an implementation for this method. Of course, the class can also include other methods and variables. For example,

```
class Line implements Drawable {
    public void draw(Graphics g) {
        . . . // do something -- presumably, draw a line
    }
}
```



```

        . . . // other methods and variables
    }

```

Any class that implements the `Drawable` interface defines a `draw()` instance method. Any object created from such a class includes a `draw()` method. We say that an object implements an interface if it belongs to a class that implements the interface. For example, any object of type `Line` implements the `Drawable` interface. Note that it is not enough for the object to include a `draw()` method. The class that it belongs to has to say that it "implements `Drawable`".

While a class can extend only one other class, it can implement any number of interfaces. In fact, a class can both extend another class and implement one or more interfaces. So, we can have things like

```

class FilledCircle extends Circle
    implements Drawable, Fillable {
    . . .
}

```

The point of all this is that, although interfaces are not classes, they are something very similar. An interface is very much like an abstract class, that is, a class that can never be used for constructing objects, but can be used as a basis for making subclasses. The subroutines in an interface are abstract methods, which must be implemented in any concrete class that implements the interface. And as with abstract classes, even though you can't construct an object from an interface, you can declare a variable whose type is given by the interface. For example, if `Drawable` is an interface, and if `Line` and `FilledCircle` are classes that implement `Drawable`, then you could say:

```

Drawable figure; // Declare a variable of type Drawable. It can
                // refer to any object that implements the
                // Drawable interface.

figure = new Line(); // figure now refers to an object of class Line
figure.draw(g);      // calls draw() method from class Line

figure = new FilledCircle(); // Now, figure refers to an object
                            // of class FilledCircle.
figure.draw(g);           // calls draw() method from class FilledCircle

```

A variable of type `Drawable` can refer to any object of any class that implements the `Drawable` interface. A statement like `figure.draw(g)`, above, is legal because `figure` is of type `Drawable`, and any `Drawable` object has a `draw()` method.

Note that a **type** is something that can be used to declare variables. A type can also be used to specify the type of a parameter in a subroutine, or the return type of a function. In Java, a type can be either a class, an interface, or one of the eight built-in primitive types. These are the only possibilities. Of these, however, only classes can be used to construct new objects.

You are not likely to need to write your own interfaces until you get to the point of writing fairly complex programs. However, there are a few interfaces that are used in important ways in Java's standard packages. You'll learn about some of these standard interfaces in the next few chapters.

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** or **inner class** is any class whose definition is inside the definition of another class. Inner classes can be either **named** or **anonymous**. I will come back to the topic of anonymous classes later in this section. A named inner class looks just like any other class, except that it is nested inside another class. (It can even contain further levels of nested classes, but you shouldn't carry these things too far.)

Like any other item in a class, a named inner class can be either static or non-static. A static nested class is part of the static structure of the containing class. It can be used inside that class to create objects in the usual way. If it has not been declared private, then it can also be used outside the containing class, but when it is used outside the class, its name must indicate its membership in the containing class. This is similar to other static components of a class: A static nested class is part of the class itself in the same way that static member variables are parts of the class itself.

For example, suppose a class named `WireFrameModel` represents a set of lines in three-dimensional space. (Such models are used to represent three-dimensional objects in graphics programs.) Suppose that the `WireFrameModel` class contains a static nested class, `Line`, that represents a single line. Then, outside of the class `WireFrameModel`, the `Line` class would be referred to as `WireFrameModel.Line`. Of course, this just follows the normal naming convention for static members of a class. The definition of the `WireFrameModel` class with its nested `Line` class would look, in outline, like this:

```
public class WireFrameModel {
    . . . // other members of the WireFrameModel class

    static public class Line {
        // Represents a line from the point (x1,y1,z1)
        // to the point (x2,y2,z2) in 3-dimensional space.
        double x1, y1, z1;
        double x2, y2, z2;
    } // end class Line

    . . . // other members of the WireFrameModel class
} // end WireFrameModel
```

Inside the `WireFrameModel` class, a `Line` object would be created with the constructor `"new Line()"`. Outside the class, `"new WireFrameModel.Line()"` would be used.

A static nested class has full access to the members of the containing class, even to the private members. 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.

When you compile the above class definition, two class files will be created. Even though the definition of `Line` is nested inside `WireFrameModel`, the compiled `Line` class is stored in a separate file. The name of the class file for `Line` will be `WireFrameModel$Line.class`.

Non-static nested classes are not, in practice, very different from static nested classes, but a non-static nested class is actually associated to an object rather than to the class in which it is nested. This can get 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 non-static nested 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 non-static nested classes. It's as if each object that belongs to the containing class has its own copy of the nested class. This copy has access to all the instance methods and instance variables of the object. Two copies of the nested class in 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 class needs to use any instance variable or instance method, make it non-static. Otherwise, it might as well be static.

From outside the containing class, a non-static nested class has to be referred to as **variableName.NestedClassName**, where `variableName` is a variable that refers to the object that contains the class. This is actually rather rare, however. A non-static nested class is generally used only inside the class in which it is nested, and there it can be referred to by its simple name.

In order to create an object that belongs to a non-static nested class, you must first have an object that belongs to the containing class. (When working inside the class, the object "this" is used implicitly.) The nested class object is permanently associated with the containing class object, and it has complete access to the members of the containing class object. Looking at an example will help, and will hopefully convince you that non-static nested classes are really very natural. Consider a class that represents poker games. This class might include a nested class to represent the players of the game. This structure of the `PokerGame` class could be:

```
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 "new `game.Player()`". Again, however, this is rather rare.) The `Player` object will have access to the `deck` and `pot` instance variables in the `PokerGame` object. Each `PokerGame` object has its own `deck` and `pot` and `Players`. `Players` of that poker game use the `deck` and `pot` for that game; `players` of another poker game use the other game's `deck` and `pot`. That's the effect of making the `Player` class non-static. This is the most natural way for players to behave. A `Player` object represents a player of one particular poker game. If `Player` were a static nested class, on the other hand, it would represent the general idea of a poker player, independent of a particular poker game.

In some cases, you might find yourself writing a nested 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 nested class**. An anonymous class is created with a variation of the `new` operator that has the form

```
new superclass-or-interface ( ) {
    methods-and-variables
```

}

This constructor defines a new class, without giving it a name, and it simultaneously creates an object that belongs to that class. This form of the `new` operator can be used in any statement where a regular "new" could be used. The intention of this expression is to create: "a new object belonging to a class that is the same as **superclass-or-interface** but with these **methods-and-variables** added." The effect is to create a uniquely customized object, just at the point in the program where you need it. Note that it is possible to base an anonymous class on an interface, rather than a class. In this case, the anonymous class must implement the interface by defining all the methods that are declared in the interface.

Anonymous classes are most often used for handling events in graphical user interfaces, and we will encounter them several times in the next two chapters. For now, we will look at one not-very-plausible example. Consider the `Drawable` interface, which is defined earlier in this section. Suppose that we want a `Drawable` object that draws a filled, red, 100-pixel square. Rather than defining a separate class and then using that class to create the object, we can use an anonymous class to create the object in one statement:

```
Drawable redSquare = new Drawable() {
    void draw(Graphics g) {
        g.setColor(Color.red);
        g.fillRect(10,10,100,100);
    }
};
```

The semicolon at the end of this statement is not part of the class definition. It's the semicolon that is required at the end of every declaration statement.

When a Java class is compiled, each anonymous nested class will produce a separate class file. If the name of the main class is `MainClass`, for example, then the names of the class files for the anonymous nested classes will be `MainClass$1.class`, `MainClass$2.class`, `MainClass$3.class`, and so on.

More about Access Modifiers

A class can be declared to be `public`. A public class can be accessed from anywhere. Certain classes have to be public. A class that defines a stand-alone application must be public, so that the system will be able to get at its `main()` routine. A class that defines an applet must be public so that it can be used by a Web browser. If a class is not declared to be `public`, then it can only be used by other classes in the same "package" as the class. Packages are discussed in [Section 4.5](#). Classes that are not explicitly declared to be in any package are put into something called the default package. All the examples in this textbook are in the default package, so they are all accessible to one another whether or not they are declared public. So, except for applications and applets, which must be `public`, it makes no practical difference whether our classes are declared to be public or not.

However, once you start writing packages, it does make a difference. A package should contain a set of related classes. Some of those classes are meant to be public, for access from outside the package. Others can be part of the internal workings of the package, and they should not be made public. A package is a kind of black box. The public classes in the package are the interface. (More exactly, the public variables and subroutines in the public classes are the interface). The non-public classes are part of the non-public implementation. Of course, all the classes in the package have unrestricted access to one another.

Following this model, I will tend to declare a class `public` if it seems like it might have some general applicability. If it is written just to play some sort of auxiliary role in a larger project, I am more likely not to make it `public`.

A member variable or subroutine in a class can also be declared to be `public`, which means that it is accessible from anywhere. It can be declared to be `private`, which means that it is accessible only from inside the class where it is defined. Making a variable `private` gives you complete control over that

variable. The only code that will ever manipulate it is the code you write in your class. This is an important kind of protection.

If no access modifier is specified for a variable or subroutine, then it is accessible from any class in the same package as the class. As with classes, in this textbook there is no practical difference between declaring a member `public` and using no access modifier at all. However, there might be stylistic reasons for preferring one over the other. And a real difference does arise once you start writing your own packages.

There is a third access modifier that can be applied to a member variable or subroutine. If it is declared to be `protected`, then it can be used in the class where it is defined and in any subclass of that class. This is obviously less restrictive than `private` and more restrictive than `public`. Classes that are written specifically to be used as a basis for making subclasses often have `protected` members. The `protected` members are there to provide a foundation for the subclasses to build on. But they are still invisible to the public at large.

Mixing Static and Non-static

Classes, as I've said, have two very distinct purposes. A class can be used to group together a set of static member variables and static member subroutines. Or it can be used as a factory for making objects. The non-static variables and subroutine definitions in the class definition specify the instance variables and methods of the objects. In most cases, a class performs one or the other of these roles, not both.

Sometimes, however, static and non-static members are mixed in a single class. In this case, the class plays a dual role. Sometimes, these roles are completely separate. It is also possible for the static and non-static parts of a class to interact. This happens when instance methods use static member variables or call static member subroutines. An instance method belongs to an object, not to the class itself, and there can be many objects with their own versions of the instance method. But there is only one copy of a static member variable. So, effectively, we have many objects sharing that one variable.

Suppose, for example, that we want to write a `PairOfDice` class that uses the `Random` class mentioned in [Section 3](#) for rolling the dice. To do this, a `PairOfDice` object needs access to an object of type `Random`. But there is no need for each `PairOfDice` object to have a separate `Random` object. (In fact, it would not even be a good idea: Because of the way random number generators work, a program should, in general, use only one source of random numbers.) A nice solution is to have a single `Random` variable as a static member of the `PairOfDice` class, so that it can be shared by all `PairOfDice` objects. For example:

```
class PairOfDice {

    private static Random randGen = new Random();
    // (Note: Assumes that java.util.Random has been imported.)

    public int die1;    // Number showing on the first die.
    public int die2;    // Number showing on the second die.

    public PairOfDice() {
        // Constructor. Creates a pair of dice that
        // initially shows random values.
        roll();
    }

    public void roll() {
        // Roll the dice by setting each of the dice to be
        // a random number between 1 and 6.
        die1 = randGen.nextInt(6) + 1;
        die2 = randGen.nextInt(6) + 1;
    }
}
```

```
} // end class PairOfDice
```

As another example, let's rewrite the `Student` class that was used in the [Section 2](#). I've added an ID for each student and a static member called `nextUniqueID`. Although there is an ID variable in each student object, there is only one `nextUniqueID` variable.

```
public class Student {

    private String name; // Student's name.
    private int ID; // Unique ID number for this student.
    public double test1, test2, test3; // Grades on three tests.

    private static int nextUniqueID = 0;
        // keep track of next available unique ID number

    Student(String theName) {
        // Constructor for Student objects;
        // provides a name for the Student,
        // and assigns the student a unique
        // ID number.
        name = theName;
        nextUniqueID++;
        ID = nextUniqueID;
    }

    public String getName() {
        // Accessor method for reading value of private
        // instance variable, name.
        return name;
    }

    public int getID() {
        // Accessor method for reading value of ID.
        return ID;
    }

    public double getAverage() {
        // Compute average test grade.
        return (test1 + test2 + test3) / 3;
    }

} // end of class Student
```

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.

End of Chapter 5

[[Next Chapter](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Programming Exercises For Chapter 5

THIS PAGE CONTAINS programming exercises based on material from [Chapter 5](#) of this [on-line Java textbook](#). Each exercise has a link to a discussion of one possible solution of that exercise.

Exercise 5.1: In all versions of the `PairOfDice` class in [Section 2](#), the instance variables `die1` and `die2` are declared to be `public`. They really should be `private`, so that they are 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 methods that can be used to find out the values of `die1` and `die2`. (The idea is to protect their values from being changed from outside the class, but still to allow the values to be read.) Include other improvements in the class, if you can think of any. Test your class with a short program that counts how many times a pair of dice is rolled, before the total of the two dice is equal to two.

[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 methods are defined:

- `calc.enter(item);` where `item` is a number, adds the item to the dataset.
- `calc.getCount()` is a function that returns the number of items that have been added to the dataset.
- `calc.getSum()` is a function that returns the sum of all the items that have been added to the dataset.
- `calc.getMean()` is a function that returns the average of all the items.
- `calc.getStandardDeviation()` is a function that returns the standard deviation of the items.

Typically, all the data are added one after the other 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 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 necessary.)

[See the solution!](#)

Exercise 5.4: The `BlackjackHand` class from [Section 5.5](#) is an extension of the `Hand` class from [Section 5.3](#). The instance methods in the `Hand` class are discussed in Section 5.3. In addition to those methods, `BlackjackHand` includes an instance method, `getBlackjackValue()`, that returns the value of the hand for the game of Blackjack. For this exercise, you will also need the `Deck` and `Card` classes from Section 5.3.

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`, your program will depend on [Card.java](#), [Deck.java](#), [Hand.java](#), and [BlackjackHand.java](#).

[See the solution!](#)

Exercise 5.5 Write a program that let's 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.

Here is an applet that simulates the program you are supposed to write. It would probably be worthwhile to play it for a while to see how it works.

**Sorry, your browser doesn't
support Java.**

[See the solution!](#)

[[Chapter Index](#) | [Main Index](#)]

Quiz Questions For Chapter 5

THIS PAGE CONTAINS A SAMPLE quiz on material from [Chapter 5](#) of this [on-line Java textbook](#). You should be able to answer these questions after studying that chapter. Sample answers to all the quiz questions can be found [here](#).

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: Explain the term *polymorphism*.

Question 8: Java uses "garbage collection" for memory management. Explain what is meant here by garbage collection. What is the alternative to garbage collection?

Question 9: 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 10: This problem uses the `Counter` class from Question 9. 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.");
```

[[Answers](#) | [Chapter Index](#) | [Main Index](#)]

Chapter 6

Applets, HTML, and GUI's

JAVA IS A PROGRAMMING LANGUAGE DESIGNED for networked computers and the World Wide Web. Java applets are downloaded over a network to appear on a Web page. Part of learning Java is learning to program applets and other Graphical User Interface programs. 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 subroutines that respond to events. Inside these subroutines, you are doing the kind of programming-in-the-small that was covered in Chapters 2 and 3. And of course, objects are everywhere. Events are objects. Applets and other GUI components are objects. Events are handled by instance methods contained in objects. In Java, event-oriented programming is object-oriented programming.

This chapter covers the basics of applets, graphics, components, and events. There is also a section that covers HyperText Markup Language (HTML), the language used for writing Web pages. The discussion of applets and GUI's will continue in the next chapter with more details and with more advanced techniques.

Contents Chapter 6:

- Section 1: [The Basic Java Applet and JApplet](#)
- Section 2: [HTML Basics and the Web](#)
- Section 3: [Graphics and Painting](#)
- Section 4: [Mouse Events](#)
- Section 5: [Keyboard Events](#)
- Section 6: [Introduction to Layouts and Components](#)
- [Programming Exercises](#)
- [Quiz on this Chapter](#)

[[First Section](#) | [Next Chapter](#) | [Previous Chapter](#) | [Main Index](#)]

Section 6.1

The Basic Java Applet and JApplet

JAVA APPLETs ARE SMALL PROGRAMS that are meant to run on a page in a Web browser. Very little of that statement is completely accurate, however. An applet is not a complete program. It doesn't have to be small. And while applets are generally meant to be used on Web pages, there are other ways to use them. A technically more correct, but not very useful, definition would say simply that an applet is an object that belongs to the class `java.applet.Applet` or to one of its subclasses. Either definition still leaves us a long way to go to really understand applets.

An applet is inherently part of a graphical user interface. It is a type of graphical component that can be displayed in a window (whether belonging to a Web browser or to some other program). When shown in a window, an applet is a rectangular area that can contain other components, such as buttons and text boxes. It can display graphical elements such as images, rectangles, and lines. And it can respond to certain "events," such as when the user clicks on the applet with a mouse.

The `Applet` class, defined in the package `java.applet`, is really only useful as a basis for making subclasses. An object of type `Applet` has certain basic behaviors, but doesn't actually do anything useful. It's just a blank area on the screen that doesn't respond to any events. To create a useful applet, a programmer must define a subclass that extends the `Applet` class. There are several methods in the `Applet` class that are defined to do nothing at all. The programmer must override at least some of these methods and give them something to do.

Back in [Section 2.1](#), when you first learned about Java programs, you encountered the idea of a `main()` routine, which is not meant to be called by the programmer. The `main()` routine of a program is there to be called by "the system" when it needs to execute the program. The programmer writes the main routine to say what happens when the system runs the program. An applet needs no `main()` routine, since it is not a stand-alone program. However, many of the methods in an applet are similar to `main()` in that they are meant to be called by the system, and the job of the programmer is to say what happens in response to the system's calls.

In this section, we'll look at a few of the things that applets can do. We'll spend the rest of this chapter and the next filling in the details.

One of the methods that is defined in the `Applet` class to do nothing is the `paint()` method. You've already encountered this method briefly in [Section 3.7](#). The `paint()` method is called by the system when the applet needs to be drawn. In a subclass of `Applet`, the `paint()` method can be redefined to draw various graphical elements such as rectangles, lines, and text on the applet. The definition of this method must have the form:

```
public void paint(Graphics g) {  
    // draw some stuff  
}
```

The parameter `g`, of type `Graphics`, is provided by the system when it calls the `paint()` method. In Java, all drawing of any kind is done using methods provided by a `Graphics` object. There are many such methods. I will discuss graphics in more detail in [Section 3](#).

As a first example of an applet, let's go the traditional route and look at an applet that displays the string "Hello World!". We'll use the `paint()` method to display this string. The `import` statements at the beginning make it possible to use the short names `Applet` and `Graphics` instead of the full names of the classes `java.applet.Applet` and `java.awt.Graphics`. (See [Section 4.5](#) for a discussion of "packages," such as `java.awt` and `java.applet`.)

```

import java.awt.*;
import java.applet.*;

public class HelloWorldApplet extends Applet {

    // An applet that simply displays the string Hello World!

    public void paint(Graphics g) {
        g.drawString("Hello World!", 10, 30);
    }

} // end of class HelloWorldApplet

```

The `drawString()` method, defined in the `Graphics` class, actually does the drawing. The parameters of this method specify the string to be drawn and the point in the applet where the string is to be placed. More about this later.

Now, an applet is an object, not a class. So far we have only defined a class. Where does an actual applet object come from? It is possible, of course, to create such objects:

```
Applet hw = new HelloWorldApplet();
```

This might even be useful if you are writing a program and would like to add an applet to a window you've created. Most often, however, applet objects are created by "the system." For example, when an applet appears on a page in a Web browser, "the system" means the Web browser. It is up to the browser program to create the applet object and to add it to a Web page. The Web browser, in turn, gets instructions about what is to appear on a given Web page from the source document for that page. For an applet to appear on a Web page, the source document for that page must specify the name of the applet and its size. This specification, like the rest of the document, is written in a language called HTML. I will discuss HTML in more detail in [Section 2](#). Here is some HTML code that instructs a Web browser to display a `HelloWorldApplet`:

```

<center>
<applet code="HelloWorldApplet.class" width=200 height=50>
</applet>
</center>

```

and here is the applet that this code displays:

(Applet "HelloWorldApplet" would be displayed here
if Java were available.)

If you are viewing this page with a web browser that supports Java, you should see the message "Hello world!". The message is displayed in a rectangle that is 200 pixels in width and 50 pixels in height. You shouldn't be able to see the rectangle as such, since by default, an applet has a background color that is the same as the color of the Web page on which it is displayed. (This might not actually be the case in your browser.)

The `Applet` class defines another method that is essential for programming applets, the `init()` method. This method is called just after the applet object has been created and before it appears on the screen. Its purpose is to give the applet a chance to do any necessary initialization. Again, this method is called by the system, not by your program. Your job as a programmer is just to provide a definition of the `init()` method. The definition of the method must have the form:

```

public void init() {
    // do initialization
}

```


(You might wonder, by the way, why initialization is done in the `init()` method rather than in a constructor. In fact, it is possible to define a constructor for your applet class. To create the applet object, the system calls the constructor that has no parameters. You can write such a constructor for an applet class and can do initializations in the constructor as well as in the `init()` method. The most significant difference is that when the constructor is called, the size of the applet is not available. By the time `init()`, is called, the size is known and can be used to customize the initialization according to the size. In general, though, it is customary to do applet initialization in the `init()` method.)

Suppose, for example, that we want to change the colors used by the `HelloWorldApplet`. An applet has a "background color" which is used to fill the entire area of the applet before any other drawing is done, and it has a "foreground color" which is used as the default color for drawing in the applet. It is convenient to set these colors in the `init()` method. Here is a version of the `HelloWorldApplet` that does this:

(Applet "HelloWorldApplet2" would be displayed here
if Java were available.)

and here is the source code for this applet, including the `init()` method:

```
import java.awt.*;
import java.applet.*;

public class HelloWorldApplet2 extends Applet {

    public void init() {
        // Initialize the applet by setting it to use blue
        // and yellow as background and foreground colors.
        setBackground(Color.blue);
        setForeground(Color.yellow);
    }

    public void paint(Graphics g) {
        g.drawString("Hello World!", 10, 30);
    }

} // end of class HelloWorldApplet2
```

JApplets and Swing

The AWT (Abstract Windowing Toolkit) has been part of Java from the beginning, but, almost from the beginning, it has been clear that the AWT was not powerful or flexible enough for writing complex, sophisticated applications. This does not prevent it from being useful -- especially for applets, which are generally not as complex as full-scale, independent applications. The Swing graphical user interface library was created to address the problems with the AWT. With the release of Java version 1.2, Swing became an official part of Java. (Versions of Java starting with 1.2 are also called, rather confusingly, "Java 2.") There are still good reasons to write applets based on the AWT, such as the lack of support in many Web browsers for Java 2. However, at this point, anyone writing a stand-alone graphical application in Java should almost certainly be using Swing, and it is Swing that I will concentrate on in this book. If you want to write applets using the AWT, you might want to look at the previous version of this book, which can be found on the web at <http://math.hws.edu/eck/cs124/javanotes3/>.

The classes that make up the Swing library can be found in the package `javax.swing`. Swing includes the class `javax.swing.JApplet` to be used as a basis for writing applets. `JApplet` is actually a subclass of `Applet`, so JApplets are in fact Applets in the usual sense. However, JApplets have a lot of extra structure that plain Applets don't have. Because of this structure, the painting of a JApplet is a more

complex affair and is handled by the system. So, when you make a subclass of `JApplet` you should **not** write a `paint()` method for it. As we will see, if you want to draw on a `JApplet`, you should add a component to the applet to be used for that purpose. On the other hand, you can and generally should write an `init()` method for a subclass of `JApplet`.

In this book, I will use a plain Applet in only a few examples. In almost all cases, I will use a `JApplet` even where a plain applet might make more sense (that is, when the applet is just being used as a simple drawing surface).

Let's take a look at a simple `JApplet` that uses Swing. This applet demonstrates some of the basic ideas of GUI programming. Although you won't understand everything in it at this time, it will give you a preliminary idea of how things work.

GUI programs use "components" such as buttons to allow interaction with the user. Our sample applet contains a button. In fact, the button is the only thing in the applet, and it fills the entire rather small applet. Here's our sample `JApplet`, which is named `HelloSwing`:

**(Applet "HelloSwing" would be displayed here
if Java were available.)**

If you click this button, a new window will open with a message and an "OK" button. Click the "OK" button to dismiss the window.

The button in this applet is an object that belongs to the class `JButton` (more properly, `javax.swing.JButton`). When the applet is created, the button must be created and added to the applet. This is part of the process of initializing the applet and is done in the applet's `init()` method. In this method, the button is created with the statement:

```
JButton btn = new JButton("Click Me!");
```

The parameter to the constructor specifies the text that is displayed on the button. The button does not automatically appear on the screen. It has to be added to the applet's "content pane." This is done with the statement:

```
getContentPane().add(btn);
```

Once it has been added to the applet, a `JButton` object mostly takes care of itself. In particular, it draws itself, so you don't have to worry about drawing it. When the user clicks the button, it generates an event. The applet (or, in fact, any object) can be programmed to respond to this event. Event-handling is the major topic in GUI programming, and I will cover it in detail later. But in outline, it works like this: The type of event generated by a button is called an `ActionEvent`. For the applet to respond to an event of this type, it must define a method

```
public void actionPerformed(ActionEvent evt) { . . . }
```

Furthermore, the button must be told that the applet will be "listening" for action events from the button. This is done by calling one of the button object's instance methods, `addActionListener()`, in the applet's `init()` method.

What should the applet do in its `actionPerformed()` method? When the user clicks the button, we want a message window to appear on the screen. Fortunately, Swing makes this easy. The class `swing.javax.JOptionPane` has a static method named `showMessageDialog()` that can be used for this purpose, so all we have to do in `actionPerformed()` is call that method.

Given all this, you can understand a lot of what goes on in the source code for the `HelloSwing` applet. This example shows several aspects of applet programming: An `init()` method sets up the applet and adds components, the components generate events, and event-handling methods say what happens in response to those events. Here is the source code:

```

// An applet that appears on the page as a button that says
// "Click Me!".  When the button is clicked, an informational
// dialog box appears to say Hello from Swing.

import javax.swing.*;    // Swing GUI classes are defined here.
import java.awt.event.*; // Event handling class are defined here.

public class HelloSwing extends JApplet implements ActionListener {

    public void init() {
        // This method is called by the system before the applet
        // appears.  It is used here to create the button and add
        // it to the "content pane" of the JApplet.  The applet
        // is also registered as an ActionListener for the button.

        JButton btnn = new JButton("Click Me!");
        btnn.addActionListener(this);
        getContentPane().add(btnn);

    } // end init()

    public void actionPerformed(ActionEvent evt) {
        // This method is called when an action event occurs.
        // In this case, the only possible source of the event
        // is the button.  So, when this method is called, we know
        // that the button has been clicked.  Respond by showing
        // an informational dialog box.  The dialog box will
        // contain an "OK" button which the user must click to
        // dismiss the dialog box.

        String title = "Greetings"; // Shown in title bar of dialog box.
        String message = "Hello from the Swing User Interface Library.";
        JOptionPane.showMessageDialog(null, message, title,
                                     JOptionPane.INFORMATION_MESSAGE);

    } // end actionPerformed()

} // end class HelloSwing

```

In this source code, I've set up the applet itself to listen for action events from the button. Some people don't consider this to be very good style. They prefer to create a separate object to listen for and respond to events. This is more "object-oriented" in the sense that each object has its own clearly defined area of responsibility. The most convenient way to make a separate event-handling object is to use a nested anonymous class. These classes were introduced in [Section 5.6](#). We will see more examples of this in the future, but here, for the record, is a version of `HelloSwing` that uses an anonymous class for event handling. This applet has exactly the same behavior as the original version:

```

import javax.swing.*;    // Swing GUI classes are defined here.
import java.awt.event.*; // Event handling class are defined here.

public class HelloSwing2 extends JApplet {

    public void init() {
        // This method is called by the system before the applet
        // appears.  It is used here to create the button and add

```

```
// it to the "content pane" of the JApplet.  An anonymous
// class is used to create an ActionListener for the button.

JButton btnn = new JButton("Click Me!");

btnn.addActionListener( new ActionListener() {
    // The "action listener" for the button is defined
    // by this nested anonymous class.
    public void actionPerformed(ActionEvent evt) {
        // This method is called to respond when the user
        // presses the button.  It displays a message in
        // a dialog box, along with an "OK" button which
        // the user can click to dismiss the dialog box.
        String title = "Greetings"; // Shown in box's title bar.
        String message = "Another hello from Swing.";
        JOptionPane.showMessageDialog(null, message, title,
                                     JOptionPane.INFORMATION_MESSAGE);
    } // end actionPerformed()
});

getContentPane().add(btnn);

} // end init()

} // end class HelloSwing2
```

[[Next Section](#) | [Previous Chapter](#) | [Chapter Index](#) | [Main Index](#)]

Section 6.2

HTML Basics

APPLETS GENERALLY APPEAR ON PAGES in a Web browser program. Such pages are themselves written in a language called **HTML** (HyperText Markup Language). An HTML document describes the contents of a page. A Web browser interprets the HTML code to determine what to display on the page. The HTML code doesn't look much like the resulting page that appears in the browser. The HTML document does contain all the text that appears on the page, but that text is "marked up" with commands that determine the structure and appearance of the text and determine what will appear on the page in addition to the text.

HTML has developed rapidly in the last few years, and it has become a rather complicated language. In this section, I will cover just the basics of the language. While that leaves out all the fancy stuff, it does include just about everything I've used to make the Web pages in this on-line text.

It is possible to write an HTML page using an ordinary text editor, typing in all the mark-up commands by hand. However, there are many Web-authoring programs that make it possible to create Web pages without ever looking at the underlying code. Using these tools, you can compose a Web page in much the same way that you would write a paper with a word processor. For example, Netscape Composer, which is part of Netscape Communicator, works in this way. However, my opinion is that making high-quality Web pages still requires some work with raw HTML, and serious Web authors still need to learn the HTML language.

The mark-up commands used by HTML are called **tags**. An HTML tag takes the form

<tag-name optional-modifiers>

Where the **tag-name** is a word that specifies the command, and the **optional-modifiers**, if present, are used to provide additional information for the command (much like parameters in subroutines). A modifier takes the form

modifier-name = value

Usually, the **value** is enclosed in quotes, and it must be if it is more than one word long or if it contains certain special characters. There are a few modifiers which have no value, in which case only the name of the modifier is present. HTML is case insensitive, which means that you can use uppercase and lowercase letters interchangeably in tags and modifiers.

A simple example of a tag is **<HR>**, which draws a line -- also called a "horizontal rule" -- across the page. The **HR** tag can take several possible modifiers such as **WIDTH** and **ALIGN**. For example, the short line just after the heading of this page was produced by the HTML command:

```
<HR align=center width="33%">
```

The **WIDTH** here is specified as 33% of the available space. It could also be given as a fixed number of pixels. The value for **ALIGN** could be **CENTER**, **LEFT**, or **RIGHT**. A **LEFT** alignment would shove the line to the left side of the page, and a **RIGHT** alignment, to the right side. **WIDTH** and **ALIGN** are optional modifiers. If you leave them out, then their **default values** will be used. The default for **WIDTH** is 100%, and the default for **ALIGN** is **LEFT**.

Many tags require matching closing tags, which take the form

</tag-name>

For example, the tag **<PRE>** must always have a matching closing tag **</PRE>** later in the document. The

tag applies to everything that comes between the opening tag and the closing tag. The `<PRE>` tag tells a Web browser to display everything between the `<PRE>` and the `</PRE>` just as it is formatted in the original HTML source code, including all the spaces and carriage returns. (But tags between `<PRE>` and `</PRE>` are still interpreted by the browser.) "PRE" stands for preformatted text. All of the sample programs in these notes are formatted using the `<PRE>` command.

It is important for you to understand that when you don't use `PRE`, the computer will completely ignore the formatting of the text in the HTML source code. The only thing it pays attention to is the tags. Five blank lines in the source code have no more effect than one blank line or even a single blank space. Outside of `<PRE>`, if you want to force a new line on the Web page, you can use the tag `
`, which stands for "break". For example, I might give my address as:

```
David Eck<BR>
Department of Mathematics and Computer Science<BR>
Hobart and William Smith Colleges<BR>
Geneva, NY 14456<BR>
```

If you want extra vertical space in your web page, you can use several `
`'s in a row.

Similarly, you need a tag to indicate how the text should be broken up into paragraphs. This is done with the `<P>` tag, which should be placed at the beginning of every paragraph. The `<P>` tag has a matching `</P>`, which should be placed at the end of each paragraph. The closing `</P>` is technically optional, but it is considered good form to use it. If you want all the lines of the paragraph to be shoved over to the right, you can use `<P ALIGN=RIGHT>` instead of `<P>`. (This is mostly useful when used with one short line, or when used with `
` to make several short lines.) You can also use `<P ALIGN=CENTER>` for centered lines.

By the way, if tags like `<P>` and `<HR>` have special meanings in HTML, you might wonder how I can get them to appear here on this page. To get certain special characters to appear on the page, you have to use an **entity name** in the HTML source code. The entity name for `<` is `<`, and the entity name for `>` is `>`. Entity names begin with `&` and end with a semicolon. The character `&` is itself a special character whose entity name is `&`. There are also entity names for nonstandard characters such as the accented e, `é`, which has the entity name `é`.

The rest of this page discusses several other basic HTML tags. This is not meant to be a complete discussion. But it is enough to produce interesting pages.

Overall Document Structure

HTML documents have a standard structure. They begin with `<HTML>` and end with `</HTML>`. Between these tags, there are two sections, the head, which is marked off by `<HEAD>` and `</HEAD>`, and the body, which -- as I'm sure you have guessed -- is surrounded by `<BODY>` and `</BODY>`. Often, the head contains only one item: a title for the document. This title might be shown, for example, in the title bar of a Web browser window. The title should not contain any HTML tags. The body contains the actual page contents that are displayed by the browser. So, an HTML document takes this form:

```
<HTML>

<HEAD>
<TITLE>page-title</TITLE>
</HEAD>

<BODY>
```


page-contents

</BODY>

</HTML>

Web browsers are not very picky about enforcing this structure; you can probably get away with leaving out everything but the actual page contents. But it is good form to follow this structure for your pages.

The <BODY> tag can take a number of modifiers that affect the appearance of the page when it is displayed. The modifier named `BGCOLOR` can be used to set the background color of the page. For example,

```
<BODY bgcolor=white>
```

will ensure that the background color for the page is white. You can add modifiers to control the color of regular text (`TEXT`), hypertext links (`LINK`), and links to pages that have already been visited (`VLINK`). When the user clicks and holds the mouse button on a link, the link is said to be active; you can control the color of active links with the `ALINK` modifier. For example, how about a page with a black background, white text, blue links, red active links, and gray visited links:

```
<BODY bgcolor=black text=white link=blue alink=red vlink=gray>
```

There are several standard color names that you can use in this context, but if you want complete control, you'll have to learn how to specify colors using hexadecimal numbers. It is also possible to use an image for the background of the page, instead of a solid color. Look up the details if you are interested.

Headings and Font Styles

HTML has a number of tags that affect the size and style of displayed text. For a heading, which is meant to stand out on a line by itself, HTML offers the tags `<H1>`, `<H2>`, ..., `<H6>`. These tags are always used with matching closing tags such as `</H1>`. The `<H1>` tag is meant for the most important headings and produces the largest size text. I've found `<H4>` through `<H6>` to be too small to be useful. You can use `
` tags in headings, if you want multi-line headings. You can also use links and images, which are described below. The heading tags can take `ALIGN` as a modifier, with the value `LEFT`, `RIGHT`, or `CENTER`. For example, the heading

A Sample Heading

was written as "`<H1 align=center>A Sample Heading</H1>`" in the HTML source code.

There are a number of different **style tags** that you can apply to text. For example, bold text can be obtained by surrounding the text with `` and ``. You can use `<i>` for italic, `<U>` for underlined, and `<TT>` for typewriter style text. Most browsers support `<SUB>` for subscripted text and `<SUP>` for superscripted text. For example, "`x²`" will give: x^2 .

Because HTML is meant to describe the logical structure of a document, rather than its exact appearance, it has a number of tags for displaying the **logical style** of the text. For example, the `` tag is meant to *emphasize* the text surrounded by `` and ``, while `` is for strong emphasis. And the `<CITE>` style tag is meant for titles of books.

You can get even more control over the style of the text by using the `...` tag. The `` tag uses modifiers such as `COLOR` and `SIZE` to control the appearance of the font. For **big blue text**, you would say:

```
<FONT color=blue size="+1">big blue text</FONT>
```

The value `" +1 "` for the `SIZE` modifier means "a little bigger than usual." You could use `" +2 "` for an even bigger font, `" -1 "` for a smaller font, and so on. However, only a limited number of different sizes are available.

Lists

There are several tags for producing lists of items. The most widely used of these are `` and ``. The `` tag gives an "ordered list", in which the items are numbered consecutively. The item numbers are provided by the browser. The `` tag gives an "unordered list", in which the items are all marked with the same special symbol. In the HTML source code, each list item is indicated by placing a `` tag at the beginning of the item. The end of the list is marked by the appropriate closing tag, `` or ``. For example, the following source code:

```
<UL>
<LI>Isaac Asimov
<LI>Ursula Leguin
<LI>Greg Bear
<LI>C. J. Cherryh
</UL>
```

produces this list:

- Isaac Asimov
- Ursula Leguin
- Greg Bear
- C. J. Cherryh

Links

The most distinctive feature of HTML is that documents can contain **links** to other documents. The user can follow links from page to page and in the process visit pages from all over the Internet.

The `<A>` tag is used to create a link. The text between the `<A>` and its matching `` appears on the page. Usually, it is underlined and in a special color. The user can follow the link by clicking on this text. The `<A>` tag uses the modifier `HREF` to say which document the link should connect to. The value for `HREF` must be a **URL** (Uniform Resource Locator). A URL is a coded set of instructions for finding a document on the Internet. For example, the URL for my own "home page" is

<http://math.hws.edu/eck/>

To make a link to this page, such as [David's Home Page](#), I would use the HTML source code

```
<A HREF="http://math.hws.edu/eck/">David's Home Page</A>
```

The best place to find URLs is on existing Web pages. Most browsers display the URL for the page you are currently viewing, and they can display the URL of a link if you point to the link with the mouse.

If you are writing an HTML document and you want to make a link to another document that is in the same directory, you can use a **relative URL**. A relative URL consists of just the name of the file. For example, the page you are now viewing comes from a directory that also contains the other sections in this chapter. For a link to [Section 1](#), which is in a file named `s1.html`, the relative URL would be just `"s1.html"`, and the complete link would look like

```
<A HREF="s1.html">Section 1</A>
```

There are also relative URLs for linking to files that are in other directories. Using relative URLs is a good idea, since if you use them, you can move a whole collection of files without changing any of the links between them (as long as you don't change the relative locations of the files).

When you type a URL into a Web browser, you can omit the `"http://"` at the beginning of the URL. However, in an `<A>` tag in an HTML document, the `"http://"` can only be omitted if the URL is a relative URL. For a normal URL, it is required.

Images

You can add images to a Web page with the `` tag. (This is a tag that has no matching closing tag.) The actual image must be stored in a separate file from the HTML document. The `` tag has a required modifier, named `SRC`, to specify the URL of the image file. For most browsers, the image should be in one of the formats GIF (with a file name ending in `".gif"`) or JPEG (with a file name ending in `".jpeg"` or `".jpg"`). A so-called **animated gif** file actually contains a series of images that the browser will display as an animation. Usually, the image is stored in the same place as the HTML document, and a relative URL is used to specify the image file.

The `` tag also has several optional modifiers. It's a good idea to always include the `HEIGHT` and `WIDTH` modifiers, which specify the size of the image in pixels. Some browsers, including Netscape, handle images better if they know in advance how big they are. For browsers that can't display images, you can use the `ALT` modifier to specify a string that will be displayed by the browser in place of the image.

The `ALIGN` modifier can be used to affect the placement of the image. `"ALIGN=RIGHT"` will shove the image to the right edge of the page, and the text on the page will flow around the image. `"ALIGN=LEFT"` works similarly. (Unfortunately, `"ALIGN=CENTER"` doesn't have the meaning you would expect. Browsers treat images as if they are just big characters. Images can occur inside paragraphs, links, and headings, for example. Alignment values of `CENTER`, `TOP`, and `BOTTOM` are used to specify how the image should line up with other characters in a line of text: Should the baseline of the text be at the center, the top, or the bottom of the image? Alignment values of `RIGHT` and `LEFT` were added to HTML later, but they are the most useful values.)

For example, here is HTML code that will place an image from a file named `figure1.gif` on the page.

```
<IMG SRC="figure1.gif" ALIGN=RIGHT HEIGHT=150
      WIDTH=100 ALT="Figure 1">
```

The image is 100 pixels wide and 150 pixels high. It will appear on the right edge of the page. If a browser can't display images, it will display the string `"Figure 1"` instead.

There are many places on the Web where you can get graphics for use on your Web pages. For example, <http://www.iconbazaar.com> makes a large number of images available. You should, of course, check on the owner's copyright policy before using someone else's images on your pages.

The Applet tag and Applet Parameters

The `<APPLET>` tag is used to add a Java applet to a Web page. This tag must have a matching `</APPLET>`. A required modifier named `CODE` gives the name of the compiled class file that contains the applet. `HEIGHT` and `WIDTH` modifiers are required to specify the size of the applet. If you want the applet to be centered on the page, you can put the applet in a paragraph with `CENTER` alignment. So, an applet tag to display an applet named `HelloWorldApplet` centered on a Web page would look like this:

```
<P ALIGN=CENTER>
  <APPLET CODE="HelloWorldApplet.class" HEIGHT=50 WIDTH=150>
</APPLET>
</P>
```

This assumes that the file `HelloWorldApplet.class` is located in the same directory with the HTML document. If this is not the case, you can use another modifier, `CODEBASE`, to give the URL of the directory that contains the class file. The value of `CODE` itself is always just a file name, not a URL.

If an applet uses a lot of `.class` files, it's a good idea to collect all the `.class` files into a single `.zip` or `.jar` file. Zip and jar files are **archive files** which hold a number of smaller files. Your Java development system is probably capable of creating them in some way. If your class files are in an archive, then you have to specify the name of the archive file in an `ARCHIVE` modifier in the `<APPLET>` tag. Archive files won't work on older browsers, but they should work for any browser that understands Java version 1.1 or later.

Applets can use **applet parameters** to customize their behavior. Applet parameters are specified by using `<PARAM>` tags, which can only occur between an `<APPLET>` tag and the closing `</APPLET>`. The `PARAM` tag has required modifiers named `NAME` and `VALUE`, and it takes the form

```
<PARAM NAME="param-name" VALUE="param-value">
```

The parameters are available to the applet when it runs. An applet can use the predefined method `getParameter()` to check for parameters specified in `PARAM` tags. The `getParameter()` method has the following interface:

```
String getParameter(String paramName)
```

The parameter `paramName` corresponds to the **param-name** in a `PARAM` tag. If the specified `paramName` actually occurs in one of the `PARAM` tags, then `getParameter` returns the associated **param-value**. If the specified `paramName` does not occur in any `PARAM` tag, then `getParameter` returns the value `null`. Parameter names are case-sensitive, so you can't use "size" in the `PARAM` tag and ask for "Size" in `getParameter`.

By the way, if you put anything besides `PARAM` tags between `<APPLET>` and `</APPLET>`, it will be ignored by any browser that supports Java. On the other hand, a browser that does not support Java will ignore the `APPLET` and `PARAM` tags. This means that if you put a message such as "Your browser doesn't support Java" between `<APPLET>` and `</APPLET>`, then that message will only appear in browsers that don't support Java.

Here is an example of an `APPLET` tag with `PARAMs` and some extra text for display in browsers that don't support Java:

```
<APPLET code="ShowMessage.class" WIDTH=200 HEIGHT=50>
  <PARAM NAME="message" VALUE="Goodbye World!">
  <PARAM NAME="font" VALUE="Serif">
  <PARAM NAME="size" VALUE="36">
  <p align=center>Sorry, but your browser doesn't support Java!</p>
</APPLET>
```

The applet ShowMessage would presumably read these parameters in its `init()` method, which might go something like this:

```
String display; // Instance variable: message to be displayed.
String fontName; // Instance variable: font to use for display.

public void init() {
    String value;
    value = getParameter("message"); // Get message PARAM, if any.
    if (value == null)
        display = "Hello World!"; // default value
    else
        display = value; // Value from PARAM tag.
    value = getParameter("font");
    if (value == null)
        fontName = "SansSerif"
    else
        fontName = value;
    .
    .
    .
}
```

Dealing with the `size` parameter would be just a little harder, since a parameter value is always a `String`, and the `size` is supposed to be an `int`. This means that the `String` value must somehow be converted to an `int`. We'll worry about how to do that later.

[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Section 6.3

Graphics and Painting

EVERYTHING YOU SEE ON A COMPUTER SCREEN has to be drawn there, even the text. The Java API includes a range of classes and methods that are devoted to drawing. In this section, I'll look at some of the most basic of these.

An applet is an example of a GUI component. The term **component** refers to a visual element in a GUI, including buttons, menus, text-input boxes, scroll bars, check boxes, and so on. In Java, GUI components are represented by objects belonging to subclasses of the class `java.awt.Component`. Most components in the Swing GUI -- although not top-level components like `JApplet` -- belong to subclasses of the class `javax.swing.JComponent`. Every component is responsible for drawing itself. For example, if you want to use a standard component, you only have to add it to your applet. You don't have to worry about painting it on the screen. That will happen automatically.

Sometimes, however, you do want to draw on a component. You will have to do this whenever you want to display something that is not included among the standard, pre-defined component classes. When you want to do this, you have to define your own component class and provide a method in that class for drawing the component.

As we have seen in [Section 6.1](#) and in [Section 3.7](#), when painting on a plain, non-Swing Applet, the drawing is done in a `paint()` method. To do custom drawing, you have to define a subclass of `Applet` and include a `paint()` method to do the drawing. However, when it comes to Swing and `JApplets`, things are a little more complicated. You should not draw directly on `JApplets` or on other top-level Swing components. Instead, you should make a separate component to use as a drawing surface, and you should add that component to the `JApplet`. You will have to write a class to represent the drawing surface, so programming a `JApplet` that does custom drawing will always involve writing at least two classes: a class for the applet itself and a class for the drawing surface. Typically, the class for the drawing surface will be defined as a subclass of `javax.swing.JPanel`, which by default is nothing but a blank area on the screen. A `JPanel`, like any `JComponent`, draws its content in the method

```
public void paintComponent(Graphics g)
```

To create a drawing surface, you should define a subclass of `JPanel` and provide a custom `paintComponent()` method. Create an object belonging to your new class, and add it to your `JApplet`. When the time comes for your component to be drawn on the screen, the system will call its `paintComponent()` to do the drawing. All this is not really as complicated as it might sound. We will go over this in more detail when the time comes.

Note that the `paintComponent()` method has a parameter of type `Graphics`. The `Graphics` object will be provided by the system when it calls your method. You need this object to do the actual drawing. To do any drawing at all in Java, you need a **graphics context**. A graphics context is an object belonging to the class, `java.awt.Graphics`. Instance methods are provided in this class for drawing shapes, text, and images. Any given `Graphics` object can draw to only one location. In this chapter, that location will always be a GUI component belonging to some subclass of `JComponent`. The `Graphics` class is an abstract class, which means that it is impossible to create a graphics context directly, with a constructor. There are actually two ways to get a graphics context for drawing on a component: First of all, of course, when the `paintComponent()` method of a component is called by the system, the parameter to that method is a graphics context for drawing on the component. Second, each component has an instance method called `getGraphics()`. This method is a function that returns a graphics context that can be used for drawing on the component outside its `paintComponent()` method. The official line is that you should **not do this, and I will avoid it for the most part. But I have found it convenient to use `getGraphics()` in some cases, since it can mean better performance for certain types of drawing. (Anyway, if the people who designed Java really didn't want us to use it, they shouldn't have made the `getGraphics()` method public!**

Most components do, in fact, do all drawing operations in their `paintComponent()` methods. What happens if, in the middle of some other method, you realize that the content of the component needs to be changed? You should not call `paintComponent()` directly to make the change; this method is meant to be called only by the system. Instead, you have to inform the system that the component needs to be redrawn, and let the system do its job by calling `paintComponent()`. You do this by calling the `repaint()` method. The method

```
public void repaint();
```

is defined in the `Component` class, and so can be used with any component. You should call `repaint()` to inform the system that the component needs to be redrawn. The `repaint()` method returns immediately, without doing any painting itself. The system will call the component's `paintComponent()` method *later*, as soon as it gets a chance to do so, after processing other pending events if there are any.

Note that the system can also call `paintComponent()` for other reasons. It is called when the component first appears on the screen. It will also be called if the component is covered up by another window and then uncovered. The system does not save a copy of the component's contents when it is covered. When it is uncovered, the component is responsible for redrawing itself. (As you will see, some of our early examples will not be able to do this correctly.)

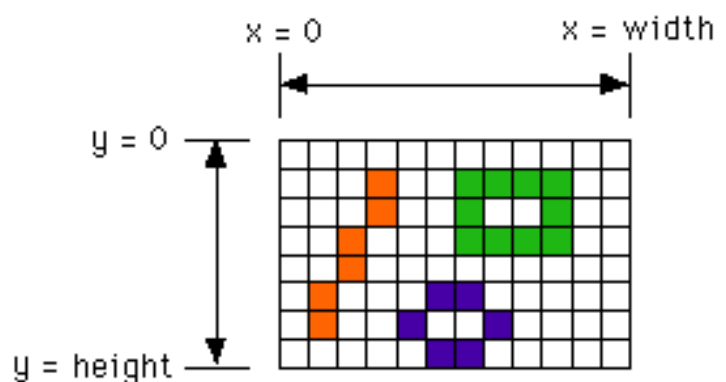
This means that, to work properly, the `paintComponent()` method must be smart enough to correctly redraw the component at any time. To make this possible, a program should store data about the state of the component in its instance variables. These variables should contain all the information necessary to redraw the component completely. The `paintComponent()` method should use the data in these variables to decide what to draw. When the program wants to change the content of the component, it should not simply draw the new content. It should change the values of the relevant variables and call `repaint()`. When the system calls `paintComponent()`, this method will use the new values of the variables and will draw the component with the desired modifications. This might seem a roundabout way of doing things. Why not just draw the modifications directly? There are at least two reasons. First of all, it really does turn out to be easier to get things right if all drawing is done in one method. Second, even if you did make modifications directly, you would still have to make the `paintComponent()` method aware of them in some way so that it will be able to redraw the component correctly when it is covered and uncovered.

You will see how all this works in practice as we work through examples in the rest of this chapter. For now, we will spend the rest of this section looking at how to get some actual drawing done.

Coordinates

The screen of a computer is a grid of little squares called **pixels**. The color of each pixel can be set individually, and drawing on the screen just means setting the colors of individual pixels.

A graphics context draws in a rectangle made up of pixels. A position in the rectangle is specified by a pair of integer coordinates, (x, y) . The upper left corner has coordinates $(0, 0)$. The x coordinate increases from left to right, and the y coordinate increases from top to bottom. The illustration on the right shows a 12-by-8 pixel component (with very large pixels). A small line, rectangle, and oval are shown as they would be drawn by coloring individual pixels. (Note that, properly speaking, the coordinates don't belong to the pixels but to the grid lines between them.)



For any component, you can find out the size of the rectangle that it occupies by calling the instance method

`getSize()`. This method returns an object that belongs to the class, `java.awt.Dimension`. A `Dimension` object has two integer instance variables, `width` and `height`. The width of the component is `getSize().width` pixels, and its height is `getSize().height` pixels.

When you are writing an applet, you don't necessarily know the applet's size. The size of an applet is usually specified in an `<APPLET>` tag in the source code of a Web page, and it's easy for the Web-page author to change the specified size. In some cases, when the applet is displayed in some other kind of window instead of on a Web page, the applet can even be resized while it is running. So, it's not good form to depend on the size of the applet being set to some particular value. For other components, you have even less chance of knowing the component's size in advance. This means that it's good form to check the size of a component before doing any drawing on that component. For example, you can use a `paintComponent()` method that looks like:

```
public void paintComponent(Graphics g) {
    int width = getSize().width;    // Find out the width of component.
    int height = getSize().height;  // Find out its height.
    . . .    // Draw the contents of the component.
}
```

Of course, your drawing commands will have to take the size into account. That is, they will have to use (x, y) coordinates that are calculated based on the actual height and width of the applet.

Colors

Java is designed to work with the **RGB color system**. An RGB color is specified by three numbers that give the level of red, green, and blue, respectively, in the color. A color in Java is an object of the class, `java.awt.Color`. You can construct a new color by specifying its red, blue, and green components. For example,

```
myColor = new Color(r,g,b);
```

There are two constructors that you can call in this way. In the one that I almost always use, `r`, `g`, and `b` are integers in the range 0 to 255. In the other, they are numbers of type `float` in the range 0.0F to 1.0F. (You might recall that a literal of type `float` is written with an "F" to distinguish it from a `double` number.) Often, you can avoid constructing new colors altogether, since the `Color` class defines several named constants representing common colors: `Color.white`, `Color.black`, `Color.red`, `Color.green`, `Color.blue`, `Color.cyan`, `Color.magenta`, `Color.yellow`, `Color.pink`, `Color.orange`, `Color.lightGray`, `Color.gray`, and `Color.darkGray`.

An alternative to RGB is the **HSB color system**. In the HSB system, a color is specified by three numbers called the **hue**, the **saturation**, and the **brightness**. The hue is the basic color, ranging from red through orange through all the other colors of the rainbow. The brightness is pretty much what it sounds like. A fully saturated color is a pure color tone. Decreasing the saturation is like mixing white or gray paint into the pure color. In Java, the hue, saturation and brightness are always specified by values of type `float` in the range from 0.0F to 1.0F. The `Color` class has a static member function named `getHSBColor` for creating HSB colors. To create the color with HSB values given by `h`, `s`, and `b`, you can say:

```
myColor = Color.getHSBColor(h,s,b);
```

For example, you could make a random color that is as bright and as saturated as possible with

```
myColor = Color.getHSBColor( (float)Math.random(), 1.0F, 1.0F );
```

The type cast is necessary because the value returned by `Math.random()` is of type `double`, and `Color.getHSBColor()` requires values of type `float`. (By the way, you might ask why RGB colors are created using a constructor while HSB colors are created using a static member function. The problem is that we would need two different constructors, both of them with three parameters of type `float`. Unfortunately, this is impossible. You can only have two constructors if the number of parameters or the parameter types

differ.)

The RGB system and the HSB system are just different ways of describing the same set of colors. It is possible to translate between one system and the other. The best way to understand the color systems is to experiment with them. In the following applet, you can use the scroll bars to control the RGB and HSB values of a color. A sample of the color is shown on the right side of the applet. Computer monitors differ as to the number of different colors they can display, so you might not get to see the full range of colors in this applet.

(Applet "ColorChooserApplet" would be displayed here
if Java were available.)

One of the instance variables in a `Graphics` object is the current drawing color, which is used for all drawing of shapes and text. If `g` is a graphics context, you can change the current drawing color for `g` using the method `g.setColor(c)`, where `c` is a `Color`. For example, if you want to draw in green, you would just say `g.setColor(Color.green)` before doing the drawing. The graphics context continues to use the color until you explicitly change it with another `setColor()` command. If you want to know what the current drawing color is, you can call the function `g.getColor()`, which returns an object of type `Color`. This can be useful if you want to change to another drawing color temporarily and then restore the previous drawing color.

Every component has an associated **foreground color** and **background color**. Generally, the component is filled with the background color before anything else is drawn (although some components are "transparent," meaning that the background color is ignored). When a new graphics context is created for a component, the current drawing color is set to the foreground color. Note that the foreground color and background color are properties of the component, not of a graphics context.

The foreground and background colors can be set by instance methods `setForeground(c)` and `setBackground(c)`, which are defined in the `Component` class and therefore are available for use with any component.

Fonts

A **font** represents a particular size and style of text. The same character will appear different in different fonts. In Java, a font is characterized by a font name, a style, and a size. The available font names are system dependent, but you can always use the following four strings as font names: "Serif", "SansSerif", "Monospaced", and "Dialog". In the original Java 1.0, the font names were "TimesRoman", "Helvetica", and "Courier". You can still use the older names if you want. (A "serif" is a little decoration on a character, such as a short horizontal line at the bottom of the letter i. "SansSerif" means "without serifs." "Monospaced" means that all the characters in the font have the same width. The "Dialog" font is the one that is typically used in dialog boxes.)

The style of a font is specified using named constants that are defined in the `Font` class. You can specify the style as one of the four values:

- `Font.PLAIN`,
- `Font.ITALIC`,
- `Font.BOLD`, or
- `Font.BOLD + Font.ITALIC`.

The size of a font is an integer. Size typically ranges from about 10 to 36, although larger sizes can also be used. The size of a font is usually about equal to the height of the largest characters in the font, in pixels, but this is not a definite rule. The size of the default font is 12.

Java uses the class named `java.awt.Font` for representing fonts. You can construct a new font by specifying its font name, style, and size in a constructor:

```
Font plainFont = new Font("Serif", Font.PLAIN, 12);
Font bigBoldFont = new Font("SansSerif", Font.BOLD, 24);
```

Every graphics context has a current font, which is used for drawing text. You can change the current font with the `setFont()` method. For example, if `g` is a graphics context and `bigBoldFont` is a font, then the command `g.setFont(bigBoldFont)` will set the current font of `g` to `bigBoldFont`. You can find out the current font of `g` by calling the method `g.getFont()`, which returns an object of type `Font`.

Every component has an associated font. It can be set with the instance method `setFont(font)`, which is defined in the `Component` class. When a graphics context is created for drawing on a component, the graphic context's current font is set equal to the font of the component.

Shapes

The `Graphics` class includes a large number of instance methods for drawing various shapes, such as lines, rectangles, and ovals. The shapes are specified using the (x, y) coordinate system described above. They are drawn in the current drawing color of the graphics context. The current drawing color is set to the foreground color of the component when the graphics context is created, but it can be changed at any time using the `setColor()` method.

Here is a list of some of the most important drawing methods. With all these commands, any drawing that is done outside the boundaries of the component is ignored. Note that all these methods are in the `Graphics` class, so they all must be called through an object of type `Graphics`.

`drawString(String str, int x, int y)` -- Draws the text given by the string `str`. The string is drawn using the current color and font of the graphics context. `x` specifies the position of the left end of the string. `y` is the y-coordinate of the baseline of the string. The baseline is a horizontal line on which the characters rest. Some parts of the characters, such as the tail on a `y` or `g`, extend below the baseline.

`drawLine(int x1, int y1, int x2, int y2)` -- Draws a line from the point (x_1, y_1) to the point (x_2, y_2) . The line is drawn as if with a pen that hangs one pixel to the right and one pixel down from the (x, y) point where the pen is located. For example, if `g` refers to an object of type `Graphics`, then the command `g.drawLine(x, y, x, y)`, which corresponds to putting the pen down at a point, draws the single pixel located at the point (x, y) .

`drawRect(int x, int y, int width, int height)` -- Draws the outline of a rectangle. The upper left corner is at (x, y) , and the width and height of the rectangle are as specified. If width equals height, then the rectangle is a square. If the width or the height is negative, then nothing is drawn. The rectangle is drawn with the same pen that is used for `drawLine()`. This means that the actual width of the rectangle as drawn is `width+1`, and similarly for the height. There is an extra pixel along the right edge and the bottom edge. For example, if you want to draw a rectangle around the edges of the component, you can say `"g.drawRect(0, 0, getSize().width-1, getSize().height-1);"`, where `g` is a graphics context for the component.

`drawOval(int x, int y, int width, int height)` -- Draws the outline of an oval. The oval is one that just fits inside the rectangle specified by `x`, `y`, `width`, and `height`. If width equals height, the oval is a circle.

`drawRoundRect(int x, int y, int width, int height, int xdiam, int ydiam)` -- Draws the outline of a rectangle with rounded corners. The basic rectangle is specified by `x`, `y`, `width`, and `height`, but the corners are rounded. The degree of rounding is given by `xdiam` and `ydiam`. The corners are arcs of an ellipse with horizontal diameter

`xdiam` and vertical diameter `ydiam`. A typical value for `xdiam` and `ydiam` is 16. But the value used should really depend on how big the rectangle is.

`draw3DRect(int x, int y, int width, int height, boolean raised)`
 -- Draws the outline of a rectangle that is supposed to have a three-dimensional effect, as if it is raised from the screen or pushed into the screen. The basic rectangle is specified by `x`, `y`, `width`, and `height`. The `raised` parameter tells whether the rectangle seems to be raised from the screen or pushed into it. The 3D effect is achieved by using brighter and darker versions of the drawing color for different edges of the rectangle. The documentation recommends setting the drawing color equal to the background color before using this method. The effect won't work well for some colors.

`drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)` -- Draws part of the oval that just fits inside the rectangle specified by `x`, `y`, `width`, and `height`. The part drawn is an arc that extends `arcAngle` degrees from a starting angle at `startAngle` degrees. Angles are measured with 0 degrees at the 3 o'clock position (the positive direction of the horizontal axis). Positive angles are measured counterclockwise from zero, and negative angles are measured clockwise. To get an arc of a circle, make sure that `width` is equal to `height`.

`fillRect(int x, int y, int width, int height)` -- Draws a filled-in rectangle. This fills in the interior of the rectangle that would be drawn by `drawRect(x,y,width,height)`. The extra pixel along the bottom and right edges is not included. The `width` and `height` parameters give the exact width and height of the rectangle. For example, if you wanted to fill in the entire component, you could say `"g.fillRect(0, 0, getSize().width, getSize().height);"`

`fillOval(int x, int y, int width, int height)` -- Draws a filled-in oval.

`fillRoundRect(int x, int y, int width, int height, int xdiam, int ydiam)` -- Draws a filled-in rounded rectangle.

`fill3DRect(int x, int y, int width, int height, boolean raised)`
 -- Draws a filled-in three-dimensional rectangle.

`fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)` -- Draw a filled-in arc. This looks like a wedge of pie, whose crust is the arc that would be drawn by the `drawArc` method.

Let's use some of the material covered in this section to write a JApplet. Since we will be drawing on the applet, we will need to create a drawing surface. The drawing surface will be a `JComponent` belonging to a subclass of the `JPanel` class. We will define this class as a nested class inside the main applet class. Nested classes were introduced in [Section 5.6](#). All the drawing is done in the `paintComponent()` method of the drawing surface class. I will use nested classes consistently to define drawing surfaces, although it is perfectly legal to use an independent class instead of a nested class to define the drawing surface. A nested class can be either static or non-static. In general, a non-static class must be used if it needs access to instance variables or instance methods that are defined in the main class. This will be the case in most of my examples.

The applet will draw multiple copies of a message on a black background. Each copy of the message is in a random color. Five different fonts are used, with different sizes and styles. The displayed message is the string "Java!", but a different message can be specified in an applet param. (Applet params were discussed at the end of the [previous section](#).) The applet works OK no matter what size is specified for the applet in the `<applet>` tag. Here's the applet:

(Applet "RandomStrings" would be displayed here
if Java were available.)

The applet does have a problem. When the drawing surface's `paintComponent()` method is called, it chooses random colors, fonts, and locations for the messages. The information about which colors, fonts, and locations are used is not stored anywhere. The next time `paintComponent()` is called, it will make different random choices and will draw a different picture. For this particular applet, the problem only really appears when the applet is *partially* covered and then uncovered. Only the part that was covered will be redrawn, and in the part that's not redrawn, the old picture will remain. Try it. You'll see partial messages, cut off by the dividing line between the new picture and the old. (Actually, in some browsers, the entire applet might be repainted, even if only part of it was covered.) A better approach would be to compute the contents of the picture elsewhere, outside the `paintComponent()` method. Information about the picture should be stored in instance variables, and the `paintComponent()` method should use that information to draw the picture. If `paintComponent()` is called twice, it should draw the same picture twice, unless the data has changed in the meantime. Unfortunately, to store the data for the picture in this applet, we would need to use either arrays, which will not be covered until [Chapter 8](#), or off-screen images, which will not be covered until [Section 7.1](#). Other applets in this chapter will suffer from the same problem.

The source for the applet is shown below. I use an instance variable called `message` to hold the message that the applet will display. There are five instance variables of type `Font` that represent different sizes and styles of text. These variables are initialized in the applet's `init()` method and are used in the drawing surface's `paintComponent()` method. I also use the `init()` method to create the drawing surface, add it to the applet, and set its background color to black.

The `paintComponent()` method for the drawing surface simply draws 25 copies of the message. For each copy, it chooses one of the five fonts at random, and it calls `g.setFont()` to select that font for drawing the text. It creates a random HSB color and uses `g.setColor()` to select that color for drawing. It then chooses random (x,y) coordinates for the location of the message. The x coordinate gives the horizontal position of the left end of the string. The formula used for the x coordinate, `"-50 + (int)(Math.random()*(width+40))"` gives a random integer in the range from `-50` to `width-10`. This makes it possible for the string to extend beyond the left edge or the right edge of the applet. Similarly, the formula for y allows the string to extend beyond the top and bottom of the applet.

The drawing surface class, which is named `Display`, defines the `paintComponent()` method that draws all the strings that appear in the applet. The drawing surface is created in the applet's `init()` method as an object of type `Display`. This object is set to be the "content pane" of the applet. A JApplet's content pane fills the entire applet, except for an optional menu bar. An applet comes with a default content pane, and you can add components to that content pane. However, any `JComponent` can be a content pane, and in a case like this where a single component fills the applet, it makes sense to replace the content pane with the `setContentPane()` method.

The `paintComponent()` method in the `Display` class begins with a call to `super.paintComponent(g)`. The special variable `super` was discussed in [Section 5.5](#). The command `super.paintComponent(g)` simply calls the `paintComponent()` method that is defined in the superclass, `JPanel`. The effect of this is to fill the component with its background color. Most `paintComponent()` methods begin with a call to `super.paintComponent(g)`, but this is not necessary if the drawing commands in the method cover the background of the component completely.

Here is the complete source code for the `RandomStrings` applet:

```
/* This applet displays 25 copies of a message. The color and
   position of each message is selected at random. The font
   of each message is randomly chosen from among five possible
   fonts. The messages are displayed on a black background.
```

```
Note: This applet uses bad style, because every time
the paintComponent() method is called, new random values are
used. This means that a different picture will be drawn each
```

time. This is particularly bad if only part of the applet needs to be redrawn, since then the applet will contain cut-off pieces of messages.

When this file is compiled, it produces two classes, RandomStrings.class and RandomStrings\$Display.class. Both classes are required to use the applet.

*/

```
import java.awt.*;
import javax.swing.*;

public class RandomStrings extends JApplet {

    String message; // The message to be displayed. This can be set in
                   // an applet param with name "message". If no
                   // value is provided in the applet tag, then
                   // the string "Java!" is used as the default.

    Font font1, font2, font3, font4, font5; // The five fonts.

    Display drawingSurface; // This is the component on which the
                           // drawing will actually be done. It
                           // is defined by a nested class that
                           // can be found below.

    public void init() {
        // Called by the system to initialize the applet.

        message = getParameter("message");
        if (message == null)
            message = "Java!";

        font1 = new Font("Serif", Font.BOLD, 14);
        font2 = new Font("SansSerif", Font.BOLD + Font.ITALIC, 24);
        font3 = new Font("Monospaced", Font.PLAIN, 20);
        font4 = new Font("Dialog", Font.PLAIN, 30);
        font5 = new Font("Serif", Font.ITALIC, 36);

        drawingSurface = new Display(); // Create the drawing surface.
        drawingSurface.setBackground(Color.black);

        setContentPane(drawingSurface); // Since drawingSurface will fill
                                       // the entire applet, we simply
                                       // replace the applet's content
                                       // pane with drawingSurface.
    } // end init()

    class Display extends JPanel {
        // This nested class defines a JPanel that is used
        // for displaying the content of the applet. An
        // object of this class is used as the content pane
        // of the applet. Note that since this is a nested
        // non-static class, it has access to the instance
        // variables of the main class such as message and font1.
    }
}
```

```

public void paintComponent(Graphics g) {

    super.paintComponent(g);    // Call the paintComponent method from
                                // the superclass, JPanel.  This simply
                                // fills the entire component with the
                                // component's background color.

    int width = getSize().width;    // Get this component's width.
    int height = getSize().height; // Get this component's 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);
                break;
            case 2:
                g.setFont(font2);
                break;
            case 3:
                g.setFont(font3);
                break;
            case 4:
                g.setFont(font4);
                break;
            case 5:
                g.setFont(font5);
                break;
        } // end switch

        // Set the color to a bright, saturated color, with random hue.

        float hue = (float)Math.random();
        g.setColor( Color.getHSBColor(hue, 1.0F, 1.0F) );

        // Select the position of the string, at random.

        int x,y;
        x = -50 + (int)(Math.random()*(width+40));
        y = (int)(Math.random()*(height+20));

        // Draw the message.

        g.drawString(message,x,y);

    } // end for

} // end paintComponent()

} // end nested class Display

```

```
} // end class RandomStrings
```

[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Section 6.4

Mouse Events

EVENTS ARE CENTRAL to programming for a graphical user interface. A GUI program doesn't have a `main()` routine that outlines what will happen when the program is run, in a step-by-step process from beginning to end. Instead, the program must be prepared to respond to various kinds of events that can happen at unpredictable times and in an order that the program doesn't control. The most basic kinds of events are generated by the mouse and keyboard. The user can press any key on the keyboard, move the mouse, or press a button on the mouse. The user can do any of these things at any time, and the computer has to respond appropriately.

In Java, events are represented by objects. When an event occurs, the system collects all the information relevant to the event and constructs an object to contain that information. Different types of events are represented by objects belonging to different classes. For example, when the user presses one of the buttons on a mouse, an object belonging to a class called `MouseEvent` is constructed. The object contains information such as the GUI component on which the user clicked, the (x, y) coordinates of the point in the component where the click occurred, and which button on the mouse was pressed. When the user presses a key on the keyboard, a `KeyEvent` is created. After the event object is constructed, it is passed as a parameter to a designated subroutine. By writing that subroutine, the programmer says what should happen when the event occurs.

As a Java programmer, you get a fairly high-level view of events. There is a lot of processing that goes on between the time that the user presses a key or moves the mouse and the time that a subroutine in your program is called to respond to the event. Fortunately, you don't need to know much about that processing. But you should understand this much: Even though your GUI program doesn't have a `main()` routine, there is a sort of main routine running somewhere that executes a loop of the form

```
while the program is still running:
    Wait for the next event to occur
    Call a subroutine to handle the event
```

This loop is called an **event loop**. Every GUI program has an event loop. In Java, you don't have to write the loop. It's part of "the system." If you write a GUI program in some other language, you might have to provide a main routine that runs an event loop.

In this section, we'll look at handling mouse events in Java, and we'll cover the framework for handling events in general. The [next section](#) will cover keyboard events. Java also has other types of events, which are produced by GUI components. These will be introduced in [Section 6](#) and covered in detail in [Section 7.3](#).

For an event to have any effect, a program must detect the event and react to it. In order to detect an event, the program must "listen" for it. Listening for events is something that is done by an object called an **event listener**. An event listener object must contain instance methods for handling the events for which it listens. For example, if an object is to serve as a listener for events of type `MouseEvent`, then it must contain the following method (among several others):

```
public void mousePressed(MouseEvent evt) { . . . }
```

The body of the method defines how the object responds when it is notified that a mouse button has been pressed. The parameter, `evt`, contains information about the event. This information can be used by the listener object to determine its response.

The methods that are required in a mouse event listener are specified in an interface named

`MouseListener`. To be used as a listener for mouse events, an object must implement this `MouseListener` interface. Java interfaces were covered in [Section 5.6](#). (To review briefly: An interface in Java is just a list of instance methods. A class can "implement" an interface by doing two things. First, the class must be declared to implement the interface, as in `class MyListener implements MouseListener` or `class RandomStrings extends JApplet implements MouseListener`. Second, the class must include a definition for each instance method specified in the interface. An interface can be used as the type for a variable or formal parameter. We say that an object implements the `MouseListener` interface if it belongs to a class that implements the `MouseListener` interface. Note that it is not enough for the object to include the specified methods. It must also belong to a class that is specifically declared to implement the interface.)

Every event in Java is associated with a GUI component. For example, when the user presses a button on the mouse, the associated component is the one that the user clicked on. Before a listener object can "hear" events associated with a given component, the listener object must be registered with the component. If a `MouseListener` object, `mListener`, needs to hear mouse events associated with a component object, `comp`, the listener must be **registered** with the component by calling `"comp.addMouseListener(mListener);"`. The `addMouseListener()` method is an instance method in the class, `Component`, and so can be used with any GUI component object. In our first few examples, we will listen for events on a `JPanel` that is being used as the drawing surface of a `JApplet`.

The event classes, such as `MouseEvent`, and the listener interfaces, such as `MouseListener`, are defined in the package `java.awt.event`. This means that if you want to work with events, you should include the line `"import java.awt.event.*;"` at the beginning of your source code file.

Admittedly, there is a large number of details to tend to when you want to use events. To summarize, you must

1. Put the import specification `"import java.awt.event.*;"` at the beginning of your source code;
2. Declare that some class implements the appropriate listener interface, such as `MouseListener`;
3. Provide definitions in that class for the subroutines from the interface;
4. Register the listener object with the component that will generate the events by calling a method such as `addMouseListener()` in the component.

Any object can act as an event listener, provided that it implements the appropriate interface. A component can listen for the events that it itself generates. An applet can listen for events from components that are contained in the applet. A special class can be created just for the purpose of defining a listening object. Many people consider it to be good form to use anonymous nested classes to define listening objects. (See [Section 5.6](#) for information on anonymous nested classes.) You will see all of these patterns in examples in this textbook.

MouseEvent and MouseListener

The `MouseListener` interface specifies five different instance methods:

```
public void mousePressed(MouseEvent evt);
public void mouseReleased(MouseEvent evt);
public void mouseClicked(MouseEvent evt);
public void mouseEntered(MouseEvent evt);
public void mouseExited(MouseEvent evt);
```

The `mousePressed` method is called as soon as the user presses down on one of the mouse buttons, and `mouseReleased` is called when the user releases a button. These are the two methods that are most commonly used, but any mouse listener object must define all five methods. You can leave the body of a method empty if you don't want to define a response. The `mouseClicked` method is called if the user

presses a mouse button and then releases it quickly, without moving the mouse. (When the user does this, all three routines -- `mousePressed`, `mouseReleased`, and `mouseClicked` -- will be called in that order.) In most cases, you should define `mousePressed` instead of `mouseClicked`. The `mouseEntered` and `mouseExited` methods are called when the mouse cursor enters or leaves the component. For example, if you want the component to change appearance whenever the user moves the mouse over the component, you could define these two methods.

As an example, let's look at an applet that does something when the user clicks on it. Here's an improved version of the `RandomStrings` applet from the end of the [previous section](#). In this version, the applet will redraw itself when you click on it:

(Applet "ClickableRandomStrings" would be displayed here
if Java were available.)

For this version of the applet, we need to make four changes in the source code. First, add the line `"import java.awt.event.*;"` before the class definition. Second, declare that some class implements the `MouseListener` interface. If we want to use the applet itself as the listener, we would do this by saying:

```
class RandomStrings extends JApplet implements MouseListener { ...
```

Third, define the five methods of the `MouseListener` interface. Only `mousePressed` will do anything. We want to repaint the drawing surface of the applet when the user clicks the mouse. The drawing surface is represented in this applet by an instance variable named `drawingSurface`, so the `mousePressed()` just needs to call `drawingSurface.repaint()` to force the drawing surface to be redrawn. The other mouse listener methods are empty. The following methods are added to the applet class definition:

```
    public void mousePressed(MouseEvent evt) {
        // When user presses the mouse, tell the system to
        // call the drawing surface's paintComponent() method.
        drawingSurface.repaint();
    }

    // The following empty routines are required by the
    // MouseListener interface:

    public void mouseEntered(MouseEvent evt) { }
    public void mouseExited(MouseEvent evt) { }
    public void mouseClicked(MouseEvent evt) { }
    public void mouseReleased(MouseEvent evt) { }
```

Fourth and finally, the applet must be registered to listen for mouse events. Since the drawing surface fills the entire applet, it is actually the drawing surface on which the user clicks. We want the applet to listen for mouse events on the drawing surface. This can be arranged by adding this line to the applet's `init()` method:

```
        drawingSurface.addMouseListener(this);
```

This calls the `addMouseListener()` method in the drawing surface object. It tells that object where to send the mouse events that it generates. The parameter to this method is the object that will be listening for the events. In this case, the listening object is the applet itself. The special variable `"this"` is used here to refer to the applet. (See [Section 5.5](#). When used in the definition of an instance method, `"this"` refers to the object that contains the method.)

We could make all these changes in the source code of the original `RandomStrings` applet. However, since we are supposed to be doing object-oriented programming, it might be instructive to write a subclass that contains the changes. This will let us build on previous work and concentrate just on the modifications.

Here's the actual source code for the above applet. It uses "super", another special variable from [Section 5.5](#).

```
import java.awt.event.*;

public class ClickableRandomStrings extends RandomStrings
    implements MouseListener {

    public void init() {
        // When the applet is created, do the initialization
        // of the superclass, RandomStrings. Then set this
        // applet to listen for mouse events on the
        // "drawingSurface". (The drawingSurface variable
        // is defined in the RandomStrings class and
        // represents a component that fills the entire applet.)
        super.init();
        drawingSurface.addMouseListener(this);
    }

    public void mousePressed(MouseEvent evt) {
        // When user presses the mouse, tell the system to
        // call the drawingSurface's paintComponent() method.
        drawingSurface.repaint();
    }

    // The next four empty routines are required by the
    // MouseListener interface.

    public void mouseEntered(MouseEvent evt) { }
    public void mouseExited(MouseEvent evt) { }
    public void mouseClicked(MouseEvent evt) { }
    public void mouseReleased(MouseEvent evt) { }

} // end class ClickableRandomStrings
```

Often, when a mouse event occurs, you want to know the location of the mouse cursor. This information is available from the parameter to the event-handling method, `evt`. This parameter is an object of type `MouseEvent`, and it contains instance methods that return information about the event. To find out the coordinates of the mouse cursor, call `evt.getX()` and `evt.getY()`. These methods return integers which give the x and y coordinates where the mouse cursor was positioned. The coordinates are expressed in the coordinate system of the component that generated the event, where the top left corner of the component is (0,0).

The user can hold down certain **modifier keys** while using the mouse. The possible modifier keys include: the Shift key, the Control key, the ALT key (called the Option key on the Macintosh), and the Meta key (called the Command or Apple key on the Macintosh and with no equivalent in Windows). You might want to respond to a mouse event differently when the user is holding down a modifier key. The boolean-valued instance methods `evt.isShiftDown()`, `evt.isControlDown()`, `evt.isAltDown()`, and `evt.isMetaDown()` can be called to test whether the modifier keys are pressed.

You might also want to have different responses depending on whether the user presses the left mouse button, the middle mouse button, or the right mouse button. Now, not every mouse has a middle button and a right button, so Java handles the information in a peculiar way. It treats pressing the right button as equivalent to holding down the Meta key while pressing the left mouse button. That is, if the right button is pressed, then the instance method `evt.isMetaDown()` will return `true` (even if the Meta key is not

pressed). Similarly, pressing the middle mouse button is equivalent to holding down the ALT key. In practice, what this really means is that pressing the right mouse button under Windows is equivalent to holding down the Command key while pressing the mouse button on Macintosh. A program tests for either of these by calling `evt.isMetaDown()`.

As an example, consider the following applet. Click on the applet with the left mouse button to place a red rectangle on the applet. Click with the right mouse button (or hold down the Command key and click on a Macintosh) to place a blue oval on the applet. Hold down the Shift key and click to clear the applet.

(Applet "SimpleStamper" would be displayed here
if Java were available.)

This applet is a `JApplet` which uses a nested class named `Display` to define its drawing surface. There are many ways to write this applet, but I decided in this case to let the drawing surface object listen for mouse events on itself. The main applet class does nothing but set up the drawing surface.

In order to respond to mouse clicks, the `Display` class implements the `MouseListener` interface, and the constructor for the display class includes the command `"addMouseListener(this)"`. Since this command is in a method in the `Display` class, the `addMouseListener()` method in the display object is being called, and `"this"` also refers to the display object. That is, the display object will send any mouse events that it generates to itself.

The source code for this applet is shown below. You can see how the instance methods in the `MouseEvent` object are used. You can also check for the Four Steps of Event Handling (`"import java.awt.event.*"`, `"implements MouseListener"`, `"addMouseListener"`, and the event-handling methods).

The `Display` class in this example violates the rule that all drawing should be done in a `paintComponent()` method. The rectangles and ovals are drawn directly in the `mousePressed()` routine. To make this possible, I need to obtain a graphics context by saying `"g = getGraphics()"`. (After using `g` for drawing, I call `g.dispose()` to inform the operating system that I will no longer be using `g` for drawing. It is a good idea to do this to free the system resources that are used by the graphics context.) I do not advise doing this type of direct drawing if it can be avoided, but you can see that it does work in this case.

By the way, this applet still has the problem that it does not save information about what has been drawn on the applet. So if the applet is covered up and uncovered, the contents of the applet are erased.

Here is the source code:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class SimpleStamper extends JApplet {

    public void init() {
        // This method is called by the system to initialize
        // the applet.  An object belonging to the nested class
        // Display is created and installed as the content
        // pane of the applet.  This Display object does
        // all the real work.
        Display display = new Display();
        setContentPane(display);
    }
}
```

```

class Display extends JPanel implements MouseListener {
    // A nested class to represent the drawing surface that
    // fills the applet.

    Display() {
        // This constructor simply sets the background color
        // of the panel to be black and sets the panel to
        // listen for mouse events on itself.
        setBackground(Color.black);
        addMouseListener(this);
    }

    public void mousePressed(MouseEvent evt) {
        // Since this panel has been set to listen for mouse
        // events on itself, this method will be called when the
        // user clicks the mouse on the panel. (Since the panel
        // fills the whole applet, that means clicking anywhere
        // on the applet.)

        if ( evt.isShiftDown() ) {
            // The user was holding down the Shift key. Just
            // repaint the panel. Since this class does not
            // define a paintComponent() method, the method
            // from the superclass, JPanel, is called. That
            // method simply fills the panel with its background
            // color, which is black. This has the effect of
            // erasing the contents of the applet.
            repaint();
            return;
        }

        int x = evt.getX(); // x-coordinate where user clicked.
        int y = evt.getY(); // y-coordinate where user clicked.

        Graphics g = getGraphics(); // Graphics context for drawing
                                    // directly on this JPanel.

        if ( evt.isMetaDown() ) {
            // User right-clicked at the point (x,y).
            // Draw a blue oval centered at the point (x,y).
            // (A black outline around the oval will make it
            // more distinct when ovals and rects overlap.)
            g.setColor(Color.blue);
            g.fillOval( x - 30, y - 15, 60, 30 );
            g.setColor(Color.black);
            g.drawOval( x - 30, y - 15, 60, 30 );
        }
        else {
            // User left-clicked (or middle-clicked) at (x,y).
            // Draw a red rectangle centered at (x,y).
            g.setColor(Color.red);
            g.fillRect( x - 30, y - 15, 60, 30 );
            g.setColor(Color.black);
            g.drawRect( x - 30, y - 15, 60, 30 );
        }
    }
}

```



```

        g.dispose(); // We are finished with the graphics context,
                    // so dispose of it.

    } // end mousePressed();

    // The next four empty routines are required by the
    // MouseListener interface.

    public void mouseEntered(MouseEvent evt) { }
    public void mouseExited(MouseEvent evt) { }
    public void mouseClicked(MouseEvent evt) { }
    public void mouseReleased(MouseEvent evt) { }

} // end nested class Display

} // end class SimpleStamper

```

MouseMotionListeners and Dragging

Whenever the mouse is moved, it generates events. The operating system of the computer detects these events and uses them to move the mouse cursor on the screen. It is also possible for a program to listen for these "mouse motion" events and respond to them. The most common reason to do so is to implement **dragging**. Dragging occurs when the user moves the mouse while holding down a mouse button.

The methods for responding to mouse motion events are defined in an interface named `MouseMotionListener`. This interface specifies two event-handling methods:

```

    public void mouseDragged(MouseEvent evt);
    public void mouseMoved(MouseEvent evt);

```

The `mouseDragged` method is called if the mouse is moved while a button on the mouse is pressed. If the mouse is moved while no mouse button is down, then `mouseMoved` is called instead. The parameter, `evt`, is an object of type `MouseEvent`. It contains the `x` and `y` coordinates of the mouse's location. As long as the user continues to move the mouse, one of these methods will be called over and over. (So many events are generated that it would be inefficient for a program to hear them all, if it doesn't want to do anything in response. This is why the mouse motion event-handlers are defined in a separate interface from the other mouse events: You can listen for the mouse events defined in `MouseListener` without automatically hearing all mouse motion events as well.)

If you want your program to respond to mouse motion events, you must create an object that implements the `MouseMotionListener` interface, and you must register that object to listen for events. The registration is done by calling a component's `addMouseMotionListener` method. The object will then listen for `mouseDragged` and `mouseMoved` events associated with that component. In most cases, the listener object will also implement the `MouseListener` interface so that it can respond to the other mouse events as well. For example, if we want an applet to listen for all mouse events associated with a `drawingSurface` object, then the definition of the applet class might have the form:

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Mouser extends JApplet

```



```

        implements MouseListener, MouseMotionListener {

    public void init() {        // set up the applet
        drasingSurface.addMouseListener(this);
        drawingSurface.addMouseMotionListener(this);
        . . . // other initializations
    }

    .
    . // Define the seven MouseListener and
    . // MouseMotionListener methods. Also, there
    . // can be other variables and methods.

```

Here is a small sample applet that displays information about mouse events. It is programmed to respond to any of the seven different kinds of mouse events by displaying the coordinates of the mouse, the type of event, and a list of the modifier keys that are down (Shift, Control, Meta, and Alt). Experiment to see what happens when you use the mouse on the applet. The source code for this applet can be found in [SimpleTrackMouse.java](#). I encourage you to read the source code. You should now be familiar with all the techniques that it uses.

(Applet "SimpleTrackMouse" would be displayed here
if Java were available.)

It is interesting to look at what a program needs to do in order to respond to dragging operations. In general, the response involves three methods: `mousePressed()`, `mouseDragged()`, and `mouseReleased()`. The dragging gesture starts when the user presses a mouse button, it continues while the mouse is dragged, and it ends when the user releases the button. This means that the programming for the response to one dragging gesture must be spread out over the three methods! Furthermore, the `mouseDragged()` method can be called many times as the mouse moves. To keep track of what is going on between one method call and the next, you need to set up some instance variables. In many applications, for example, in order to process a `mouseDragged` event, you need to remember the previous coordinates of the mouse. You can store this information in two instance variables `prevX` and `prevY` of type `int`. I also suggest having a boolean variable, `dragging`, which is set to `true` while a dragging gesture is being processed. This is necessary because not every `mousePressed` event is the beginning of a dragging gesture. The `mouseDragged` and `mouseReleased` methods can use the value of `dragging` to check whether a drag operation is actually in progress. You might need other instance variables as well, but in general outline, the code for handling dragging looks like this:

```

private int prevX, prevY; // Most recently processed mouse coords.
private boolean dragging; // Set to true when dragging is in process.
. . . // other instance variables for use in dragging

public void mousePressed(MouseEvent evt) {
    if ( we-want-to-start-dragging ) {
        dragging = true;
        prevX = evt.getX(); // Remember starting position.
        prevY = evt.getY();
    }
    .
    . // Other processing.
    .
}

public void mouseDragged(MouseEvent evt) {
    if ( dragging == false ) // First, check if we are
        return;             // processing a dragging gesture.
}

```

```

        int x = evt.getX(); // Current position of Mouse.
        int y = evt.getY();
        .
        . // Process a mouse movement from (prevX, prevY) to (x,y).
        .
        prevX = x; // Remember the current position for the next call.
        prevY = y;
    }

    public void mouseReleased(MouseEvent evt) {
        if ( dragging == false ) // First, check if we are
            return; // processing a dragging gesture.
        dragging = false; // We are done dragging.
        .
        . // Other processing and clean-up.
        .
    }
}

```

As an example, let's look at a typical use of dragging: allowing the user to sketch a curve by dragging the mouse. This example also shows many other features of graphics and mouse processing. In the following applet, you can draw a curve by dragging the mouse on the large white area. Select a color for drawing by clicking on one of the colored rectangles on the right. Note that the selected color is framed with a white border. Clear your drawing by clicking in the square labeled "CLEAR". (This applet still has the old problem that the drawing will disappear if you cover the applet and uncover it.)

(Applet "SimplePaint" would be displayed here
if Java were available.)

You'll find the complete source code for this applet in the file [SimplePaint.java](#). I will discuss a few aspects of it here, but I encourage you to read it carefully in its entirety. There are lots of informative comments in the source code. (This is actually an old-style non-Swing Applet which uses a `paint()` method to draw on the applet instead of a `paintComponent()` method to draw on a drawing surface.)

The applet class for this example is designed to work for any reasonable applet size, that is, unless the applet is too small. This means that coordinates are computed in terms of the actual width and height of the applet. (The width and height are obtained by calling `getSize().width` and `getSize().height`.) This makes things quite a bit harder than they would be if we assumed some particular fixed size for the applet. Let's look at some of these computations in detail. For example, the command used to fill in the large white drawing area is

```
g.fillRect(3, 3, width - 59, height - 6);
```

There is a 3-pixel border along each edge, so the height of the drawing area is 6 less than the height of the applet. As for the width: The colored rectangles are 50 pixels wide. There is a 3-pixel border on each edge of the applet. And there is a 3-pixel divider between the drawing area and the colored rectangles. All that adds up to make 59 pixels that are not included in the width of the drawing area, so the width of the drawing area is 59 less than the width of the applet.

The white square labeled "CLEAR" occupies a 50-by-50 pixel region beneath the colored rectangles. Allowing for this square, we can figure out how much vertical space is available for the seven colored rectangles, and then divide that space by 7 to get the vertical space available for each rectangle. This quantity is represented by a variable, `colorSpace`. Out of this space, 3 pixels are used as spacing between the rectangles, so the height of each rectangle is `colorSpace - 3`. The top of the N-th rectangle is located $(N * \text{colorSpace} + 3)$ pixels down from the top of the applet, assuming that we start counting at 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 applet. 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 applet 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` routine has to look at the (x, y) coordinates where the mouse was clicked and decide how to respond. If the user clicked on the CLEAR rectangle, the drawing area is cleared by calling `repaint()`. If the user clicked somewhere in the strip of colored rectangles, the selected color is changed. This involves computing which color the user clicked on, which is done by dividing the y coordinate by `colorSpace`. Finally, if the user clicked on the drawing area, a drag operation is initiated. 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 applet. This is not automatic, since as far as the computer is concerned, the border and the color bar are part of the drawing surface. If the user drags the mouse outside the drawing area while drawing a line, the `mouseDragged` routine changes the x and y coordinates to make them lie within the drawing area.

Anonymous Event Handlers and Adapter Classes

As I mentioned above, it is a fairly common practice to use anonymous nested classes to define listener objects. As discussed in [Section 5.6](#), a special form of the `new` operator is used to create an object that belongs to an anonymous class. For example, a mouse listener object can be created with an expression of the form:

```
new MouseListener() {
    public void mousePressed(MouseEvent evt) { . . . }
    public void mouseReleased(MouseEvent evt) { . . . }
    public void mouseClicked(MouseEvent evt) { . . . }
    public void mouseEntered(MouseEvent evt) { . . . }
    public void mouseExited(MouseEvent evt) { . . . }
}
```

This is all just one long expression that both defines an un-named class and creates an object that belongs to that class. To use the object as a mouse listener, it should be passed as the parameter to some component's `addMouseListener()` method in a command of the form:

```
component.addMouseListener( new MouseListener() {
    public void mousePressed(MouseEvent evt) { . . . }
    public void mouseReleased(MouseEvent evt) { . . . }
    public void mouseClicked(MouseEvent evt) { . . . }
    public void mouseEntered(MouseEvent evt) { . . . }
    public void mouseExited(MouseEvent evt) { . . . }
} );
```

Now, in a typical application, most of the method definitions in this class will be empty. A class that implements an interface must provide definitions for all the methods in that interface, even if the definitions are empty. To avoid the tedium of writing empty method definitions in cases like this, Java provides **adapter classes**. An adapter class implements a listener interface by providing empty definitions for all the methods in the interface. An adapter class is only useful as a basis for making subclasses. In the subclass, you can define just those methods that you actually want to use. For the remaining methods, the empty definitions that are provided by the adapter class will be used. The adapter class for the `MouseListener` interface is named `MouseAdapter`. For example, if you want a mouse listener that

only responds to mouse-pressed events, you can use a command of the form:

```
component.addMouseListener( new MouseAdapter() {
    public void mousePressed(MouseEvent evt) { . . . }
} );
```

To see how this works in a real example, let's write another version of the ClickableRandomStrings applet that uses an anonymous class based on MouseAdapter to handle mouse events:

```
import java.awt.event.*;

public class ClickableRandomStrings2 extends RandomStrings {

    public void init() {
        // When the applet is created, do the initialization
        // of the superclass, RandomStrings. Then add a
        // mouse listener to listen for mouse events on the
        // "drawingSurface". (drawingSurface is defined
        // in the superclass, RandomStrings.)

        super.init();

        drawingSurface.addMouseListener( new MouseAdapter() {
            public void mousePressed(MouseEvent evt) {
                // When user presses the mouse, tell the system to
                // call the drawingSurface's paintComponent() method.
                drawingSurface.repaint();
            }
        } );
    } // end init()

} // end class ClickableRandomStrings2
```

[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Section 6.5

Keyboard Events

IN JAVA, EVENTS are associated with GUI components. When the user presses a button on the mouse, the event that is generated is associated with the component that contains the mouse cursor. What about keyboard events? When the user presses a key, what component is associated with the key event that is generated?

A GUI uses the idea of **input focus** to determine the component associated with keyboard events. At any given time, exactly one interface element on the screen has the input focus, and that is where all keyboard events are directed. If the interface element happens to be a Java component, then the information about the keyboard event becomes a Java object of type `KeyEvent`, and it is delivered to any listener objects that are listening for `KeyEvents` associated with that component. The necessity of managing input focus adds an extra twist to working with keyboard events.

It's a good idea to give the user some visual feedback about which component has the input focus. For example, if the component is the typing area of a word-processor, the feedback is usually in the form of a blinking text cursor. Another common visual clue is to draw a brightly colored border around the edge of a component when it has the input focus, as I do in the sample applet later on this page.

A component that wants to have the input focus can call the method `requestFocus()`, which is defined in the `Component` class. Calling this method does not absolutely guarantee that the component will actually get the input focus. Several components might request the focus; only one will get it. This method should only be used in certain circumstances in any case, since it can be a rude surprise to the user to have the focus suddenly pulled away from a component that the user is working with. In a typical user interface, the user can choose to give the focus to a component by clicking on that component with the mouse. And pressing the tab key will often move the focus from one component to another.

Some components do not automatically receive the input focus when the user clicks on them. To solve this problem, a program has to register a mouse listener with the component to detect user clicks. In response to a user click, the `mousePressed()` method should call `requestFocus()` for the component. This is true, in particular, for the components that are used as drawing surfaces in the examples in this chapter. These components are defined as subclasses of `JPanel`, and `JPanel` objects do not receive the input focus automatically. If you want to be able to use the keyboard to interact with a `JPanel` named `drawingSurface`, you have to register a listener to listen for mouse events on the `drawingSurface` and call `drawingSurface.requestFocus()` in the `mousePressed()` method of the listener object.

Here is a sample applet that processes keyboard events. If the applet has the input focus, the arrow keys can be used to move the colored square. Furthermore, pressing the 'R', 'G', 'B', or 'K' key will set the color of the square to red, green, blue, or black. When the applet has the input focus, the border of the applet is a bright cyan (blue-green) color. When the applet does not have the focus, the border is gray, and a message, "Click to activate," is displayed. When the user clicks on an unfocused applet, it requests the input focus. (In some browsers, you also have to leave the mouse positioned inside the applet, in order for it to have the input focus.) The complete source code for this applet is in the file [KeyboardAndFocusDemo.java](#). I will discuss some aspects of it below. After reading this section, you should be able to understand the source code in its entirety.

(Applet "KeyboardAndFocusDemo" would be displayed here
if Java were available.)

In Java, keyboard event objects belong to a class called `KeyEvent`. An object that needs to listen for `KeyEvents` must implement the interface named `KeyListener`. Furthermore, the object must be

registered with a component by calling the component's `addKeyListener()` method. The registration is done with the command `component.addKeyListener(listener);` where `listener` is the object that is to listen for key events, and `component` is the object that will generate the key events (when it has the input focus). It is possible for `component` and `listener` to be the same object. All this is, of course, directly analogous to what you learned about mouse events in the [previous section](#). The `KeyListener` interface defines the following methods, which must be included in any class that implements `KeyListener`:

```
public void keyPressed(KeyEvent evt);
public void keyReleased(KeyEvent evt);
public void keyTyped(KeyEvent evt);
```

Java makes a careful distinction between *the keys that you press* and *the characters that you type*. There are lots of keys on a keyboard: letter keys, number keys, modifier keys such as Control and Shift, arrow keys, page up and page down keys, keypad keys, function keys. In many cases, pressing a key does not type a character. On the other hand, typing a character sometimes involves pressing several keys. For example, to type an uppercase 'A', you have to press the Shift key and then press the A key before releasing the Shift key. On my Macintosh computer, I can type an accented e, é, by holding down the Option key, pressing the E key, releasing the Option key, and pressing E again. Only one character was typed, but I had to perform three key-presses and I had to release a key at the right time. In Java, there are three types of `KeyEvent`. The types correspond to pressing a key, releasing a key, and typing a character. The `keyPressed` method is called when the user presses a key, the `keyReleased` method is called when the user releases a key, and the `keyTyped` method is called when the user types a character. 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' could generate two `keyPressed`, two `keyReleased`, 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 have a solitaire game on my computer that hilites every card that can be moved, when I hold down the Shift key. You could do something like that in Java by hiliting the cards when the Shift key is pressed and removing the hilite when the Shift key is released.

There is one more complication. Usually, when you hold down a key on the keyboard, that key will **auto-repeat**. This means that it will generate multiple `keyPressed` events, as long as it is held down. It can also generate multiple `keyTyped` events. For the most part, this will not affect your programming, but you should not expect every `keyPressed` event to have a corresponding `keyReleased` event.

Every key on the keyboard has an integer code number. (Actually, this is only true for keys that Java knows about. Many keyboards have extra keys that can't be used with Java.) When the `keyPressed` or `keyReleased` method is called, the parameter, `evt`, contains the code of the key that was pressed or released. The code can be obtained by calling the function `evt.getKeyCode()`. Rather than asking you to memorize a table of code numbers (which can be different for different platforms in any case), Java provides a named constant for each key. These constants are defined in the `KeyEvent` class. For example the constant for the shift key is `KeyEvent.VK_SHIFT`. If you want to test whether the key that the user pressed is the Shift key, you could say `"if (evt.getKeyCode() == KeyEvent.VK_SHIFT)"`. The key codes for the four arrow keys are `KeyEvent.VK_LEFT`, `KeyEvent.VK_RIGHT`, `KeyEvent.VK_UP`, and `KeyEvent.VK_DOWN`. Other keys have similar codes. (The "VK" stands for "Virtual Keyboard". In reality, different keyboards use different key codes, but Java translates the actual codes from the keyboard into its own "virtual" codes. Your program only sees these virtual key codes, so it will work with various keyboards on various platforms without modification.)

In the case of a `keyTyped` event, you want to know which character was typed. This information can be obtained from the parameter, `evt`, in the `keyTyped` method by calling the function `evt.getKeyChar()`. This function returns a value of type `char` representing the character that was typed.

In the `KeyboardAndFocusDemo` applet, shown above, I use the `keyPressed` routine to respond when the user presses one of the arrow keys. The applet includes instance variables, `squareLeft` and `squareTop` that give the position of the upper left corner of the square. When the user presses one of the arrow keys, the `keyPressed` routine modifies the appropriate instance variable and calls `canvas.repaint()` to redraw the whole applet. ("canvas" is the name I use for the drawing surface component in this applet.) Note that the values of `squareLeft` and `squareRight` are restricted so that the square never moves outside the white area of the applet:

```
public void keyPressed(KeyEvent evt) {
    // Called when the user has pressed a key, which can be
    // a special key such as an arrow key.  If the key pressed
    // was one of the arrow keys, move the square (but make sure
    // that it doesn't move off the edge, allowing for a
    // 3-pixel border all around the applet).  SQUARE_SIZE is
    // a named constant that specifies the size of the square.
    // squareLeft and squareRight give the position of the
    // top-left corner of the square.

    int key = evt.getKeyCode();  // Keyboard code for the pressed key.

    if (key == KeyEvent.VK_LEFT) {  // left-arrow key; move square up
        squareLeft -= 8;
        if (squareLeft < 3)
            squareLeft = 3;
        canvas.repaint();
    }
    else if (key == KeyEvent.VK_RIGHT) {  // right-arrow key; move right
        squareLeft += 8;
        if (squareLeft > getSize().width - 3 - SQUARE_SIZE)
            squareLeft = getSize().width - 3 - SQUARE_SIZE;
        canvas.repaint();
    }
    else if (key == KeyEvent.VK_UP) {  // up-arrow key; move up
        squareTop -= 8;
        if (squareTop < 3)
            squareTop = 3;
        canvas.repaint();
    }
    else if (key == KeyEvent.VK_DOWN) {  // down-arrow key; move down
        squareTop += 8;
        if (squareTop > getSize().height - 3 - SQUARE_SIZE)
            squareTop = getSize().height - 3 - SQUARE_SIZE;
        canvas.repaint();
    }
}  // end keyPressed()
```

Color changes -- which happen when the user types the characters 'R', 'G', 'B', and 'K', or the lower case equivalents -- are handled in the `keyTyped` method. I won't include it here, since it is so similar to the

`keyPressed` method. Finally, to complete the `KeyListener` interface, the `keyReleased` method must be defined. In the sample applet, the body of this method is empty since the applet does nothing to respond to `keyReleased` events.

Focus Events

If a component is to change its appearance when it has the input focus, it needs some way to know when it has the focus. In Java, objects are notified about changes of input focus by events of type `FocusEvent`. An object that wants to be notified of changes in focus can implement the `FocusListener` interface. This interface declares two methods:

```
public void focusGained(FocusEvent evt);
public void focusLost(FocusEvent evt);
```

Furthermore, the `addFocusListener()` method must be used to set up a listener for the focus events. When a component gets the input focus, it calls the `focusGained()` method of any object that has been registered with that component as a `FocusListener`. When it loses the focus, it calls the listener's `focusLost()` method. Often, it is the component itself that listens for focus events.

In my sample applet, there is a boolean-valued instance variable named `focussed`. This variable is `true` when the applet has the input focus and is `false` when the applet does not have focus. The applet implements the `FocusListener` interface and listens for focus events from the canvas. The `paintComponent()` method of the canvas looks at the value of `focussed` to decide what color the border should be. The value of `focussed` is set in the `focusGained()` and `focusLost()` methods. These methods call `canvas.repaint()` so that the drawing surface will be redrawn with the correct border color. The method definitions are very simple:

```
public void focusGained(FocusEvent evt) {
    // The canvas now has the input focus.
    focussed = true;
    canvas.repaint(); // redraw with cyan border
}

public void focusLost(FocusEvent evt) {
    // The canvas has now lost the input focus.
    focussed = false;
    canvas.repaint(); // redraw with gray border
}
```

The other aspect of handling focus is to make sure that the canvas gets the focus when the user clicks on it. To do this, the applet implements the `MouseListener` interface and listens for mouse events on the canvas. It defines a `mousePressed` routine that asks that the input focus be given to the canvas:

```
public void mousePressed(MouseEvent evt) {
    canvas.requestFocus();
}
```

The other four methods of the `MouseListener` interface are defined to be empty. Note that the applet implements three listener interfaces, so the class definition begins:

```
public class KeyboardAndFocusDemo extends JApplet
    implements KeyListener, FocusListener, MouseListener
```

The applet's `init()` method registers the applet to listen for all three types of events. To do this, the `init()` method includes the lines

```

canvas.addFocusListener(this);
canvas.addKeyListener(this);
canvas.addMouseListener(this);

```

There are, of course, other ways to organize this applet. It would be possible, for example, to use the `canvas` object instead of the applet object to listen for events. Or anonymous classes could be used to define separate listening objects.

State Machines

The information stored in an object's instance variables is said to represent the **state** of that object. When one of the object's methods is called, the action taken by the object can depend on its state. (Or, in the terminology we have been using, the definition of the method can look at the instance variables to decide what to do.) Furthermore, the state can change. (That is, the definition of the method can assign new values to the instance variables.) In computer science, there is the idea of a **state machine**, which is just something that has a state and can change state in response to events or inputs. The response of a state machine to an event or input depends on what state it's in. An object is a kind of state machine. Sometimes, this point of view can be very useful in designing classes.

The state machine point of view can be especially useful in the type of event-oriented programming that is required by graphical user interfaces. When designing an applet, you can ask yourself: What information about state do I need to keep track of? What events can change the state of the applet? How will my response to a given event depend on the current state? Should the appearance of the applet be changed to reflect a change in state? How should the `paintComponent()` method take the state into account? All this is an alternative to the top-down, step-wise-refinement style of program design, which does not apply to the overall design of an event-oriented program.

In the `KeyboardAndFocusDemo` applet, shown above, the state of the applet is recorded in the instance variables `focussed`, `squareLeft`, and `squareTop`. These state variables are used in the `paintComponent()` method to decide how to draw the applet. They are set in the various event-handling methods.

In the rest of this section, we'll look at another example, where the state of the applet plays an even bigger role. In this example, the user plays a simple arcade-style game by pressing the arrow keys. The example is based on one of my frameworks, called `KeyboardAnimationApplet2`. (See [Section 3.7](#) for a discussion of frameworks and a sample framework that supports animation.) The game is written as an extension of the `KeyboardAnimationApplet2` class. It includes a method, `drawFrame()`, that draws one frame in the animation. It also defines `keyPressed` to respond when the user presses the arrow keys. The source code for the game is in the file [SubKillerGame.java](#). You can also look at the source code in [KeyboardAnimationApplet2.java](#), but it uses some advanced techniques that I haven't covered yet.

You have to click on the game to activate it. The applet shows a black "submarine" moving back and forth erratically near the bottom. Near the top, there is a blue "boat". You can move this boat back and forth by pressing the left and right arrow keys. Attached to the boat is a red "depth charge." You can drop the depth charge by hitting the down arrow key. The objective is to blow up the submarine by hitting it with the depth charge. If the depth charge falls off the bottom of the screen, you get a new one. If the sub explodes, a new sub is created and you get a new depth charge. Try it! Make sure to hit the sub at least once, so you can see the explosion.

(Applet "SubKillerGame" would be displayed here
if Java were available.)

Let's think about how this applet can be programmed. What constitutes the "state" of the applet? That is, what things change from time to time and affect the appearance or behavior of the applet? Of course, the state includes the positions of the boat, submarine, and depth charge, so I need instance variables to store

the positions. Anything else, possibly less obvious? Well, sometimes the depth charge 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, `bombIsFalling`. Sometimes the submarine is moving left and sometimes it is moving right. The difference is represented by another boolean variable, `subIsMovingLeft`. Sometimes, the sub is exploding. This is also part of the state, but representing it requires a little more thought. While an explosion is in progress, the sub looks different in each frame, since the size of the explosion increases. Also, I need to know when the explosion is over so that I can go back to drawing the sub as usual. So, I use a variable, `explosionFrameNumber`, of type `int`, which tells how many frames have been drawn since the explosion started. I represent the fact that no explosion is happening by setting the value of `explosionFrameNumber` to zero. Alternatively, I could have used another boolean variable to keep track of whether or not an explosion is in progress.

How and when do the values of these instance variables change? Some of them can change when the user presses certain keys. In the program, this is checked in the `keyPressed()` method. 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 depth charge changes from not-falling to falling. This is coded as follows:

```
public void keyPressed(KeyEvent evt) {

    int code = evt.getKeyCode(); // which key was pressed

    if (code == KeyEvent.VK_LEFT) {
        // Move the boat left.
        boatCenterX -= 15;
    }
    else if (code == KeyEvent.VK_RIGHT) {
        // Move the boat right.
        boatCenterX += 15;
    }
    else if (code == KeyEvent.VK_DOWN) {
        // Start the bomb falling, if it is not already falling.
        if ( bombIsFalling == false )
            bombIsFalling = true;
    }

} // end keyPressed()
```

Note that it's not necessary to call `repaint()` when the state changes, since this applet is an animation that is constantly being redrawn anyway. Any changes in the state will become visible to the user as soon as the next frame is drawn. At some point in the program, I have to make sure that the user does not move the boat off the screen. I could have done this in `keyPressed()`, but I choose to check for this in another routine, just before drawing the boat.

Other aspects of the state are changed in the `drawFrame()` routine. From the point of view of programming, this method is handling an event ("Hey, it's time to draw the next frame!"). It just happens to be an event that is generated by the `KeyboardAnimationApplet2` framework rather than by the user. In my applet, the `drawFrame()` routine calls three other methods that I wrote to organize the process of computing and drawing a new frame: `doBombFrame()`, `doBoatFrame()`, and `doSubFrame()`.

Consider `doBombFrame()`. This routine draws the depth charge. What happens in this routine depends on the current state, and the routine can make changes to the state when it is executed. The state of the bomb can be falling or not-falling, as recorded in the variable, `bombIsFalling`. If `bombIsFalling` is false, then the bomb is simply drawn at the bottom of the boat. If `bombIsFalling` is true, the vertical coordinate of the bomb has to be increased by some amount to make the bomb move down a bit from one frame to the next. Several other things can also happen. If the bomb has fallen off the bottom of the applet

-- something that we can test by looking at its vertical coordinate -- then `bombIsFalling` becomes false. This puts the bomb back at the boat in the next frame. Also, the bomb might hit the sub. This can be tested by comparing the locations of the bomb and the sub. If the bomb hits the sub, then the state changes in two ways: the bomb is no longer falling and the sub is exploding. These state changes are implemented by setting `bombIsFalling` to false and `explosionFrameNumber` to 1.

Most interesting is the submarine. What happens with the submarine depends on whether it is exploding or not. If it is (that is, if `explosionFrameNumber > 0`), then yellow and red ovals are drawn at the sub's position. The sizes of these ovals depend on the value of `explosionFrameNumber`, so they grow with each frame of the explosion. After the ovals are drawn, the value of `explosionFrameNumber` is incremented. If its value has reached 14, it is reset to 0. This reflects a change of state: The sub is no longer exploding. It's important for you to understand what is happening here. There is no loop in the program to draw the stages of the explosion. Each frame is a new event and is drawn separately, based on values stored in instance variables. The state can change, which will make the next frame look different from the current one.

In a frame where the sub is not exploding, it moves left or right. This is accomplished by adding or subtracting a small amount to the horizontal coordinate of the sub. Whether it moves left or right is determined by the value of the variable, `subIsMovingLeft`. It's interesting to consider how and when this variable changes value. If the sub reaches the left edge of the applet, `subIsMovingLeft` is set to false to make the sub start moving right. Similarly, if the sub reaches the right edge. But the sub can also reverse direction at random times. The way this is implemented is that in each frame, there is a small chance that the sub will reverse direction. This is done with the statement

```
if ( Math.random() < 0.04 )
    subIsMovingLeft = !subIsMovingLeft;
```

Since `Math.random()` is between 0 and 1, the condition "`Math.random() < 0.04`" has a 4 in 100, or 1 in 25, chance of being true. In those frames where this condition happens to evaluate to true, the sub reverses direction. (The value of the expression "`!subIsMovingLeft`" is false when `subIsMovingLeft` is true, and it is true when `subIsMovingLeft` is false, so it effectively reverses the value of `subIsMovingLeft`.)

While it's not very sophisticated as arcade games go, the `SubKillerGame` applet does use some interesting programming. And it nicely illustrates how to apply state-machine thinking in event-oriented programming.

[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Section 6.6

Introduction to Layouts and Components

IN PRECEDING SECTIONS, YOU'VE SEEN how to use a graphics context to draw on the screen and how to handle mouse events and keyboard events. In one sense, that's all there is to GUI programming. If you're willing to program all the drawing and handle all the mouse and keyboard events, you have nothing more to learn. However, you would either be doing a lot more work than you need to do, or you would be limiting yourself to very simple user interfaces. A typical user interface uses standard GUI components such as buttons, scroll bars, text-input boxes, and menus. These components have already been written for you, so you don't have to duplicate the work involved in developing them. They know how to draw themselves, and they can handle the details of processing the mouse and keyboard events that concern them.

Consider one of the simplest user interface components, a push button. The button has a border, and it displays some text. This text can be changed. Sometimes the button is disabled, so that clicking on it doesn't have any effect. When it is disabled, its appearance changes. When the user clicks on the push button, the button changes appearance while the mouse button is pressed and changes back when the mouse button is released. In fact, it's more complicated than that. If the user moves the mouse outside the push button before releasing the mouse button, the button changes to its regular appearance. To implement this, it is necessary to respond to mouse exit or mouse drag events. Furthermore, on many platforms, a button can receive the input focus. The button changes appearance when it has the focus. If the button has the focus and the user presses the space bar, the button is triggered. This means that the button must respond to keyboard and focus events as well.

Fortunately, you don't have to program *any* of this, provided you use an object belonging to the standard class `javax.swing.JButton`. A `JButton` object draws itself and processes mouse, keyboard, and focus events on its own. You only hear from the `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 `JButton` object creates an event object belonging to the class `java.awt.event.ActionEvent`. The event object is sent to any registered listeners to tell them that the button has been pushed. Your program gets only the information it needs -- the fact that a button was pushed.

Another aspect of GUI programming is laying out components on the screen, that is, deciding where they are drawn and how big they are. You have probably noticed that computing coordinates can be a difficult problem, especially if you don't assume a fixed size for the applet. 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. An applet's content pane is an example of a container. The standard class `JPanel`, which we have only used as a drawing surface up till now, is another example of a container. Because a `JPanel` object is a container, it can hold other components. So `JPanel`s are dual purpose: You can draw on them, and you can add other components to them. Because a `JPanel` is itself a component, you can add a `JPanel` to an applet's content pane or even to another `JPanel`. This makes complex nesting of components possible. `JPanel`s can be used to organize complicated user interfaces.

The components in a container must be "laid out," which means setting their sizes and positions. It's possible to program the layout yourself, but ordinarily layout is done by a **layout manager**. A layout manager is an object associated with a container that implements some policy for laying out the components in that container. Different types of layout manager implement different policies.

In this section, we'll look at a few examples of using components and layout managers, leaving the details until [Section 7.2](#) and [Section 7.3](#). The applets that we look at in this section have a large drawing area with a row of controls below it.

Our first example is rather simple. It's another "Hello World" applet, in which the color of the message can be changed by clicking one of the buttons at the bottom of the applet:

**(Applet "HelloWorldJApplet" would be displayed here
if Java were available.)**

In the previous JApplets that we've looked at, the entire applet was filled with a JPanel that served as a drawing surface. In this example, there are two JPanels: the large black area at the top that displays the message and the smaller area at the bottom that holds the three buttons.

Let's first consider the panel that contains the buttons. This panel is created in the applet's `init()` method as a variable named `buttonBar`, of type `JPanel`:

```
JPanel buttonBar = new JPanel();
```

When a panel is to be used as a drawing surface, it is necessary to create a subclass of the `JPanel` class and include a `paintComponent()` method to do the drawing. However, when a `JPanel` is just being used as a container, there is no need to create a subclass. A standard `JPanel` is already capable of holding components of any type.

Once the panel has been created, the three buttons are created and are added to the panel. A button is just an object belonging to the class `javax.swing.JButton`. When a button is created, the text that will be shown on the button is provided as a parameter to the constructor. The first button in the panel is created with the command:

```
JButton redButton = new JButton("Red");
```

This button is added to the `buttonBar` panel with the command:

```
buttonBar.add(redButton);
```

Every `JPanel` comes automatically with a layout manager. This default layout manager will simply line up the components that are added to it in a row. That's exactly the behavior we want here, so there is nothing more to do. If we wanted a different kind of layout, it's possible to change the panel's layout manager.

One more step is required to make the button useful: an object must be registered with the button to listen for `ActionEvents`. The button will generate an `ActionEvent` when the user clicks on it. `ActionEvents` are similar to `MouseEvent`s or `KeyEvent`s. To use them, a class should import `java.awt.event.*`. The object that is to do the listening must implement an interface named `ActionListener`. This interface requires a definition for the method `"public void actionPerformed(ActionEvent evt);"`. Finally, the listener must be registered with the button by calling the button's `addActionListener()` method. In this case, the applet itself will act as listener, and the registration is done with the command:

```
redButton.addActionListener(this);
```

After doing the same three commands for each of the other two buttons -- and setting the background color for the sake of aesthetics -- the `buttonBar` panel is ready to use. It just has to be added to the applet.

As we have seen, components are not added directly to an applet. Instead, they are added to the applet's content pane, which is itself a container. The content pane comes with a default layout manager that is capable of displaying up to five components. Four of these components are placed along the edges of the applet, in the so-called "North", "South", "East", and "West" positions. A component in the "Center" position fills in all the remaining space. This type of layout is called a `BorderLayout`. In our example, the button bar occupies the "South" position and the drawing area fills the "Center" position. When you add a component to a `BorderLayout`, you have to specify its position using a constant such as `BorderLayout.SOUTH` or `BorderLayout.CENTER`. In this example, `buttonBar` is added to the applet with the command:

```
getContentPane().add(buttonBar, BorderLayout.SOUTH);
```

The display area of the applet is a drawing surface like those we have seen in other examples. A nested class named `Display` is created as a subclass of `JPanel`, and the display area is created as an object belonging to that class. The applet class has an instance variable named `display` of type `Display` to represent the drawing surface. The display object is simply created and added to the applet with the commands:

```
display = new Display();
getContentPane().add(display, BorderLayout.CENTER);
```

Putting this all together, the complete `init()` method for the applet becomes:

```
public void init() {

    display = new Display();
        // The component that displays "Hello World".

    getContentPane().add(display, BorderLayout.CENTER);
        // Adds the display panel to the CENTER position of the
        // JApplet's content pane.

    JPanel buttonBar = new JPanel();
        // This panel will hold three buttons and will appear
        // at the bottom of the applet.

    buttonBar.setBackground(Color.gray);
        // Change the background color of the button panel
        // so that the buttons will stand out better.

    JButton redButton = new JButton("Red");
        // Create a new button. "Red" is the text
        // displayed on the button.

    redButton.addActionListener(this);
        // Set up the button to send an "action event" to this applet
        // when the user clicks the button. The parameter, this,
        // is a name for the applet object that we are creating,
        // so action events from the button will be handled by
        // calling the actionPerformed() method in this class.

    buttonBar.add(redButton);
        // Add the button to the buttonBar panel.

    JButton greenButton = new JButton("Green"); // the second button
    greenButton.addActionListener(this);
    buttonBar.add(greenButton);

    JButton blueButton = new JButton("Blue"); // the third button
    blueButton.addActionListener(this);
    buttonBar.add(blueButton);

    getContentPane().add(buttonBar, BorderLayout.SOUTH);
        // Add button panel to the bottom of the content pane.

} // end init()
```

Notice that the variables `buttonBar`, `redButton`, `greenButton`, and `blueButton` are local to the

`init()` method. This is because once the buttons and panel have been added to the applet, the variables are no longer needed. The objects continue to exist, since they have been added to the applet. But they will take care of themselves, and there is no need to manipulate them elsewhere in the applet. The `display` variable, on the other hand, is an instance variable that can be used throughout the applet. This is because we are *not* finished with the `display` object after adding it to the applet. When the user clicks a button, we have to change the color of the display. We need a way to keep the variable around so that we can refer to it in the `actionPerformed()` method. In general, you don't need an instance variable for every component in an applet -- just for the components that will be referred to outside the `init()` method.

The drawing surface in our example is defined by a nested class named `Display` which is a subclass of `JPanel`. The class contains a `paintComponent()` method that is responsible for drawing the message "Hello World" on a black background. The `Display` class also contains a variable that it uses to remember the current color of the message and a method that can be called to change the color. This class is more self-contained than most of the drawing surface classes that we have looked at, and in fact it could have been defined as an independent class instead of as a nested class. Here is the definition of the nested class, `Display`:

```
class Display extends JPanel {
    // This nested class defines a component that displays
    // the string "Hello World". The color and font for
    // the string are recorded in the variables colorNum
    // and textFont.

    int colorNum;        // Keeps track of which color is displayed;
                        //      1 for red, 2 for green, 3 for blue.

    Font textFont;       // The font in which the message is displayed.
                        // A font object represents a certain size and
                        // style of text drawn on the screen.

    Display() {
        // Constructor for the Display class. Set the background
        // color and assign initial values to the instance
        // variables, colorNum and textFont.
        setBackground(Color.black);
        colorNum = 1;    // The color of the message is set to red.
        textFont = new Font("Serif",Font.BOLD,36);
        // Create a font object representing a big, bold font.
    }

    void setColor(int code) {
        // This method is provided to be called by the
        // main class when it wants to set the color of the
        // message. The parameter value should be 1, 2, or 3
        // to indicate the desired color.
        colorNum = code;
        repaint(); // Tell the system to repaint this component.
    }

    public void paintComponent(Graphics g){
        // This routine is called by the system whenever this
        // panel needs to be drawn or redrawn. It first calls
        // super.paintComponent() to fill the panel with the
        // background color. It then displays the message
```

```

        // "Hello World" in the proper color and font.
        super.paintComponent(g);
        switch (colorNum) {           // Set the color.
            case 1:
                g.setColor(Color.red);
                break;
            case 2:
                g.setColor(Color.green);
                break;
            case 3:
                g.setColor(Color.blue);
                break;
        }
        g.setFont(textFont);          // Set the font.
        g.drawString("Hello World!", 25,50);    // Draw the message.
    } // end paintComponent

} // end nested class Display

```

The main class has an instance variable named `display` of type `Display`. When the user clicks one of the buttons in the applet, this variable is used to call the `setColor()` method in the drawing surface object. This is done in the applet's `actionPerformed()` method. This method is called when the user clicks any one of the three buttons, so it needs some way to tell which button was pressed. This information is provided in the parameter to the `actionPerformed()` method. This parameter contains an "action command," which in the case of a button is just the string that is displayed on the button:

```

public void actionPerformed(ActionEvent evt) {
    // This routine is called by the system when the user clicks
    // on one of the buttons. The response is to set the display's
    // color accordingly.

    String command = evt.getActionCommand();
        // The "action command" associated with the event
        // is the text on the button that was clicked.

    if (command.equals("Red"))           // Set the color.
        display.setColor(1);
    else if (command.equals("Green"))
        display.setColor(2);
    else if (command.equals("Blue"))
        display.setColor(3);

} // end actionPerformed()

```

We have now looked at all the pieces of the sample applet. You can find the entire source code in the file [HelloWorldJApplet.java](#).

For a second example, let's look at something a little more interesting. Here's a simple card game in which you look at a playing card and try to predict whether the next card will be higher or lower in value. (Aces have the lowest value in this game.) You've seen a text-oriented version of the same game in [Section 5.3](#). That section also defined `Deck`, `Hand`, and `Card` classes that are used in this applet. In this GUI version of the game, you click on a button to make your prediction. If you predict wrong, you lose. If you make three

correct predictions, you win. After completing one game, you can click the "New Game" button to start a new game. Try it! See what happens if you click on one of the buttons at a time when it doesn't make sense to do so.

(Applet "HighLowGUI" would be displayed here
if Java were available.)

The overall form of this applet is the same as that of the previous example: It has three buttons in a panel at the bottom of the applet and a large drawing surface that displays the cards and a message. However, I've organized the code a little differently in this example. In this case, it's the drawing surface object, rather than the applet, that listens for events from the buttons, and I've put almost all the programming into the display surface class. The applet object is only responsible for creating the components and adding them to the applet. This is done in the following `init()` method, which has almost the same form as the `init()` method in the previous example:

```
public void init() {

    // The init() method lays out the applet. A HighLowCanvas
    // occupies the CENTER position of the layout. On the
    // bottom is a panel that holds three buttons. The
    // HighLowCanvas object listens for ActionEvents from the
    // buttons and does all the real work of the program.

    setBackground( new Color(130,50,40) );

    HighLowCanvas board = new HighLowCanvas();
    getContentPane().add(board, BorderLayout.CENTER);

    JPanel buttonPanel = new JPanel();
    buttonPanel.setBackground( new Color(220,200,180) );
    getContentPane().add(buttonPanel, BorderLayout.SOUTH);

    JButton higher = new JButton( "Higher" );
    higher.addActionListener(board);
    buttonPanel.add(higher);

    JButton lower = new JButton( "Lower" );
    lower.addActionListener(board);
    buttonPanel.add(lower);

    JButton newGame = new JButton( "New Game" );
    newGame.addActionListener(board);
    buttonPanel.add(newGame);

} // end init()
```

In programming the drawing surface class, `HighLowCanvas`, 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 [Section 5.3](#) 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 canvas class uses a boolean variable named `gameInProgress` for this purpose.

The state of the applet can change whenever the user clicks on a button. The `HighLowCanvas` class implements the `ActionListener` interface and defines an `actionPerformed()` method to respond to the user's clicks. This method simply calls one of three other methods, `doHigher()`, `doLower()`, or `newGame()`, depending on which button was pressed. It's in these three event-handling methods that the action of the game takes place.

We don't want to let the user start a new game if a game is currently in progress. That would be cheating. So, the response in the `newGame()` method is different depending on whether the state variable `gameInProgress` is true or false. If a game is in progress, the message instance variable should be set to show an error message. If a game is not in progress, then all the state variables should be set to appropriate values for the beginning of a new game. In any case, the board must be repainted so that the user can see that the state has changed. The complete `newGame()` method is as follows:

```
void doNewGame() {
    // Called by the constructor, and called by actionPerformed()
    // when the user clicks the "New Game" button. Start a new game.
    if (gameInProgress) {
        // If the current game is not over, it is an error to try
        // to start a new game.
        message = "You still have to finish this game!";
        repaint();
        return;
    }
    deck = new Deck(); // Create a deck and hand to use for this game.
    hand = new Hand();
    deck.shuffle();
    hand.addCard( deck.dealCard() ); // Deal the first card.
    message = "Is the next card higher or lower?";
    gameInProgress = true; // State changes! A game has started.
    repaint();
}
```

The `doHigher()` and `doLower()` methods are almost identical to each other (and could probably have been combined into one method with a parameter, if I were more clever). Let's look at the `doHigher()` routine. This is called when the user clicks the "Higher" button. This only makes sense if a game is in progress, so the first thing `doHigher()` should do is check the value of the state variable `gameInProgress`. If the value is false, then `doHigher()` should just set up an error message. If a game is in progress, a new card should be added to the hand and the user's prediction should be tested. The user might win or lose at this time. If so, the value of the state variable `gameInProgress` must be set to false because the game is over. In any case, the board is repainted to show the new state. Here is the `doHigher()` method:

```
void doHigher() {
    // Called by actionPerformed() when user clicks "Higher".
    // Check the user's prediction. Game ends if user guessed
    // wrong or if the user has made three correct predictions.
    if (gameInProgress == false) {
        // If the game has ended, it was an error to click "Higher",
        // so set up an error message and abort processing.
        message = "Click \"New Game\" to start a new game!";
        repaint();
        return;
    }
}
```

```

        hand.addCard( deck.dealCard() );           // Deal a card to the hand.
        int cardCt = hand.getCardCount();          // How many cards in the hand?
        Card thisCard = hand.getCard( cardCt - 1 ); // Card just dealt.
        Card prevCard = hand.getCard( cardCt - 2 ); // The previous card.
        if ( thisCard.getValue() < prevCard.getValue() ) {
            gameInProgress = false;
            message = "Too bad! You lose.";
        }
        else if ( thisCard.getValue() == prevCard.getValue() ) {
            gameInProgress = false;
            message = "Too bad! You lose on ties.";
        }
        else if ( cardCt == 4 ) {
            gameInProgress = false;
            message = "You win! You made three correct guesses.";
        }
        else {
            message = "Got it right! Try for " + cardCt + ".";
        }
        repaint();
    }

```

The `paintComponent()` method of the `HighLowCanvas` class uses the values in the state variables to decide what to show. It displays the string stored in the `message` variable. It draws each of the cards in the hand. There is one little tricky bit: If a game is in progress, it draws an extra face-down card, which is not in the hand, to represent the next card in the deck. Drawing the cards requires some care and computation. I wrote a method, "`void drawCard(Graphics g, Card card, int x, int y)`", which draws a card with its upper left corner at the point (x, y) . The `paintComponent()` routine decides where to draw each card and calls this routine to do the drawing. You can check out all the details in the source code, [HighLowGUI.java](#).

As a final example, let's look quickly at an improved paint program, similar to the one from [Section 4](#). The user can draw on the large white area. In this version, the user selects the drawing color from the pop-up menu at the bottom-left of the applet. If the user hits the "Clear" button, the drawing area is filled with the background color. I've added one feature: If the user hits the "Set Background" button, the background color of the drawing area is set to the color currently selected in the pop-up menu, and the drawing area is cleared. This lets you draw in cyan on a magenta background if you have a mind to.

**(Applet "SimplePaint2" would be displayed here
if Java were available.)**

The drawing area in this applet is a component, belonging to the nested class `SimplePaintCanvas`. I wrote this class, as usual, as a sub-class of `JPanel` and programmed it to listen for mouse events and to respond by drawing a curve. As in the `HighLowGUI` applet, all the action takes place in the nested class. The main applet class just does the set up. One new feature of interest is the pop-up menu. This component is an object belonging to the standard class, `JComboBox`. We'll cover this component class in Chapter 7.

What you should note about this version of the paint applet is that in many ways, it was easier to write than the original. There are no computations about where to draw things and how to decode user mouse clicks. We don't have to worry about the user drawing outside the drawing area. The graphics context that is used for drawing on the canvas can only draw on the canvas. If the user tries to extend a curve outside the canvas, the part that lies outside the canvas is automatically ignored. We don't have to worry about giving the user visual feedback about which color is selected. That is handled by the text displayed on the pop-up menu.

You'll find the source code for this example in the file [SimplePaint2.java](#). After struggling through this chapter, you should be equipped to understand it almost in its entirety!

End of Chapter 6

[[Next Chapter](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Programming Exercises For Chapter 6

THIS PAGE CONTAINS programming exercises based on material from [Chapter 6](#) of this [on-line Java textbook](#). Each exercise has a link to a discussion of one possible solution of that exercise.

Exercise 6.1: Write an applet that shows a pair of dice. When the user clicks on the applet, the dice should be rolled (that is, the dice should be assigned newly computed random values). Each die should be drawn as a square showing from 1 to 6 dots. Since you have to draw two dice, its a good idea to write a subroutine, `"void drawDie(Graphics g, int val, int x, int y)"`, to draw a die at the specified (x,y) coordinates. The second parameter, val, specifies the value that is showing on the die. Assume that the size of the applet is 100 by 100 pixels. Here is a working version of the applet. (My applet plays a clicking sound when the dice are rolled. See the solution to see how this is done.)

[See the solution!](#)

Exercise 6.2: Improve your dice applet from the previous exercise so that it also responds to keyboard input. When the applet has the input focus, it should be hilited with a colored border, and the dice should be rolled whenever the user presses a key on the keyboard. This is in addition to rolling them when the user clicks the mouse on the applet. Here is an applet that solves this exercise:

[See the solution!](#)

Exercise 6.3: In Exercise 6.1, above, you wrote a pair-of-dice applet where the dice are rolled when the clicks on the applet. Now make a pair-of-dice applet that uses the methods discussed in [Section 6.6](#). Draw the dice on a JPanel, and place a "Roll" button at the bottom of the applet. The dice should be rolled when the user clicks the Roll button. Your applet should look and work like this one:

(Note: Since there was only one button in this applet, I added it directly to the applet's content pane, rather than putting it in a "buttonBar" panel and adding the panel to the content pane.)

[See the solution!](#)

Exercise 6.4: In [Exercise 3.5](#), you drew a checkerboard. For this exercise, write a checkerboard applet where the user can select a square by clicking on it. Hilite the selected square by drawing a colored border around it. When the applet is first created, no square is selected. When the user clicks on a square that is not currently selected, it becomes selected. If the user clicks the square that is selected, it becomes unselected. Assume that the size of the applet is 160 by 160 pixels, so that each square on the checkerboard is 20 by 20 pixels. Here is a working version of the applet:

[See the solution!](#)

Exercise 6.5: Write an applet that shows two squares. The user should be able to drag either square with the mouse. (You'll need an instance variable to remember which square the user is dragging.) The user can drag the square off the applet if she wants; if she does this, it's gone. You can try it here:

[See the solution!](#)

Exercise 6.6: For this exercise, you should modify the SubKiller game from [Section 6.5](#). You can start with the existing source code, from the file [SubKillerGame.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 applet 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!](#) (A working version of the applet can be found [here](#).)

Exercise 6.7: [Section 3.7](#) discussed `SimpleAnimationApplet2`, a framework for writing simple animations. You can define an animation by writing a subclass and defining a `drawFrame()` method. It is possible to have the subclass implement the `MouseListener` interface. Then, you can have an animation that responds to mouse clicks.

Write a game in which the user tries to click on a little square that jumps erratically around the applet. To implement this, use instance variables to keep track of the position of the square. In the `drawFrame()` method, there should be a certain probability that the square will jump to a new location. (You can experiment to find a probability that makes the game play well.) In your `mousePressed` method, check whether the user clicked on the square. Keep track of and display the number of times that the user hits the square and the number of times that the user misses it. Don't assume that you know the size of the applet in advance.

[See the solution!](#) (A working version of the applet can be found [here](#).)

Exercise 6.8: Write a Blackjack applet that lets the user play a game of Blackjack, with the computer as the dealer. The applet should draw the user's cards and the dealer's cards, just as was done for the graphical HighLow card game in [Section 6.6](#). You can use the source code for that game, [HighLowGUI.java](#), for some ideas about how to write your Blackjack game. The structures of the HighLow applet and the Blackjack applet are very similar. You will certainly want to use the `drawCard()` method from that applet.

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 applet, it means that you only have to allow room for five cards. You should assume that your applet is just wide enough to show five cards, and that it is tall enough to show the user's hand and the dealer's hand.

Note that the design of a GUI Blackjack game is very different from the design of the text-oriented program that you wrote for Exercise 5.5. The user should play the game by clicking on "Hit" and "Stand" buttons. There should be a "New Game" button that can be used to start another game after one game ends. You have to decide what happens when each of these buttons is pressed. You don't have much chance of getting this right unless you think in terms of the states that the game can be in and how the state can change.

Your program will need the classes defined in [Card.java](#), [Hand.java](#), [BlackjackHand.java](#), and [Deck.java](#). Here is a working version of the applet:

[See the solution!](#)

Quiz Questions For Chapter 6

THIS PAGE CONTAINS A SAMPLE quiz on material from [Chapter 6](#) of this [on-line Java textbook](#). You should be able to answer these questions after studying that chapter. Sample answers to all the quiz questions can be found [here](#).

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: What is an *event loop*?

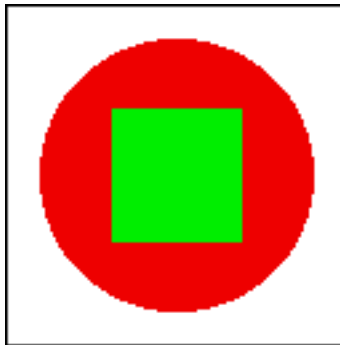
Question 3: Explain carefully what the `repaint()` method does.

Question 4: What is HTML?

Question 5: Draw the picture that will be produced by the following `paint()` method:

```
public static void paint(Graphics g) {
    for (int i=10; i <= 210; i = i + 50)
        for (int j = 10; j <= 210; j = j + 50)
            g.drawLine(i,10,j,60);
}
```

Question 6: Suppose you would like an applet that displays a green square inside a red circle, as illustrated. Write a `paint()` method that will draw the image.



Question 7: Suppose that you are writing an applet, and you want the applet to respond in some way when the user clicks the mouse on the applet. What are the four things you need to remember to put into the source code of your applet?

Question 8: Java has a standard class called `MouseEvent`. What is the purpose of this class? What does an object of type `MouseEvent` do?

Question 9: Explain what is meant by *input focus*. How is the input focus managed in a Java GUI program?

Question 10: Java has a standard class called `JPanel`. Discuss *two* ways in which `JPanels` can be used.

[[Answers](#) | [Chapter Index](#) | [Main Index](#)]

Chapter 7

Advanced GUI Programming

IT'S POSSIBLE TO PROGRAM A WIDE VARIETY of GUI applications using only the techniques covered in the previous chapter. In many cases, the basic events, components, layouts, and graphics routines covered in that chapter suffice. But the Swing graphical user interface library is far richer than what we have seen so far, and it can be used to build highly sophisticated applications. This chapter is a further introduction to Swing. Although the title of the chapter is "Advanced GUI Programming," it is still just an introduction. Full coverage of Swing would require at least another complete book.

In this chapter, we'll take a more detailed look at Swing, starting with a few more features of the `Graphics` class. We'll cover a number of new layout managers, component classes, and event types, and we'll see how to open independent windows and dialog boxes on the screen. We'll also look at two other programming techniques, timers and threads.

The material in this chapter will be used in a number of examples and programming exercises in future chapters. Aside from that, this chapter is not a prerequisite the rest of this textbook. If you skip it, you will not miss out on any fundamental programming concepts -- just a lot of the fun of GUI programming.

Contents Chapter 7:

- Section 1: [More about Graphics](#)
- Section 2: [More about Layouts and Components](#)
- Section 3: [Basic Components and Their Events](#)
- Section 4: [Programming with Components](#)
- Section 5: [Menus and Menubars](#)
- Section 6: [Timers, Animation, and Threads](#)
- Section 7: [Frames and Applications](#)
- [Programming Exercises](#)
- [Quiz on this Chapter](#)

[[First Section](#) | [Next Chapter](#) | [Previous Chapter](#) | [Main Index](#)]

Section 7.1

More About Graphics

IN THIS SECTION, we'll look at some additional aspects of graphics in Java. Most of the section deals with `Images`, which are pictures stored in files or in the computer's memory. But we'll also consider a few other techniques that can be used to draw better or more efficiently.

Images

To a computer, an image is just a set of numbers. The numbers specify the color of each pixel in the image. The numbers that represent the image on the computer's screen are stored in a part of memory called a **frame buffer**. Many times each second, the computer's video card reads the data in the frame buffer and colors each pixel on the screen according to that data. Whenever the computer needs to make some change to the screen, it writes some new numbers to the frame buffer, and the change appears on the screen a fraction of a second later, the next time the screen is redrawn by the video card.

Since it's just a set of numbers, the data for an image doesn't have to be stored in a frame buffer. It can be stored elsewhere in the computer's memory. It can be stored in a file on the computer's hard disk. Just like any other data file, an image file can be downloaded over the Internet. Java includes standard classes and subroutines that can be used to copy image data from one part of memory to another and to get data from an image file and use it to display the image on the screen.

The standard class `java.awt.Image` is used to represent images. A particular object of type `Image` contains information about some particular image. There are actually two kinds of `Image` objects. One kind represents an image in an image data file. The second kind represents an image in the computer's memory. Either type of image can be displayed on the screen. The second kind of `Image` can also be modified while it is in memory. We'll look at this second kind of `Image` below.

Every image is coded as a set of numbers, but there are various ways in which the coding can be done. For images in files, there are two main coding schemes which are used in Java and on the Internet. One is used for GIF images, which are usually stored in files that have names ending in ".gif". The other is used for JPEG images, which are stored in files that have names ending in ".jpg" or ".jpeg". Both GIF and JPEG images are **compressed**. That is, redundancies in the data are exploited to reduce the number of numbers needed to represent the data. In general, the compression method used for GIF images works well for line drawings and other images with large patches of uniform color. JPEG compression generally works well for photographs.

The `Applet` class defines a method, `getImage`, that can be used for loading images stored in GIF and JPEG files. (As we will see later, stand-alone applications use a different technique for loading image files.) For example, suppose that the image of an ace of clubs, shown at the right, is contained in a file named "ace.gif". And suppose that `img` is a variable of type `Image`. Then the following command could be used in the source code of your applet:



```
img = getImage( getCodeBase(), "ace.gif" );
```

This would create an `Image` object to represent the ace. The second parameter is the name of the file that contains the image. The first parameter specifies the directory that contains the image file. The value "`getCodeBase()`" specifies that the image file is in the code base directory for the applet. Assuming that the applet is in the default package, as usual, that just means that the image file is in the same directory as the compiled class file of the applet.

Once you have an object of type `Image`, however you obtain it, you can draw the image in any graphics context. Most commonly, this will be done in the `paintComponent()` method of a `JPanel` (or some other `JComponent`.) If `g` is the `Graphics` object that is provided as a parameter to the `paintComponent()` method, then the command:

```
g.drawImage(img, x, y, this);
```

will draw the image `img` in a rectangular area in the component. The parameters `x` and `y` give the position of the upper-left corner of the rectangle in which the image is displayed, and the rectangle is just large enough to hold the image. The fourth parameter, `this`, is the special variable from [Section 5.5](#) that refers to the component itself. This parameter is there for technical reasons having to do with the funny way Java treats image files. (Although you don't really need to know this, here is how it works: When you use `getImage()` to create an `Image` object from an image file, the file is not downloaded immediately. The `Image` object simply remembers where the file is. The file will be downloaded the first time you draw the image. However, when the image needs to be downloaded, the `drawImage()` method only initiates the downloading. It doesn't wait for the data to arrive. So, after `drawImage()` has finished executing, it's quite possible that the image has not actually been drawn! But then, when does it get drawn? That's where the fourth parameter to the `drawImage()` command comes in. The fourth parameter is something called an `ImageObserver`. After the image has been downloaded, the system will inform the `ImageObserver` that the image is available, and the `ImageObserver` will actually draw the image at that time. For large images, it's even possible that the image will be drawn in several parts as it is downloaded. Any `JComponent` object can act as an `ImageObserver`. If you are sure that the image that you are drawing has already been downloaded, you can set the fourth parameter of `drawImage()` to `null`.)

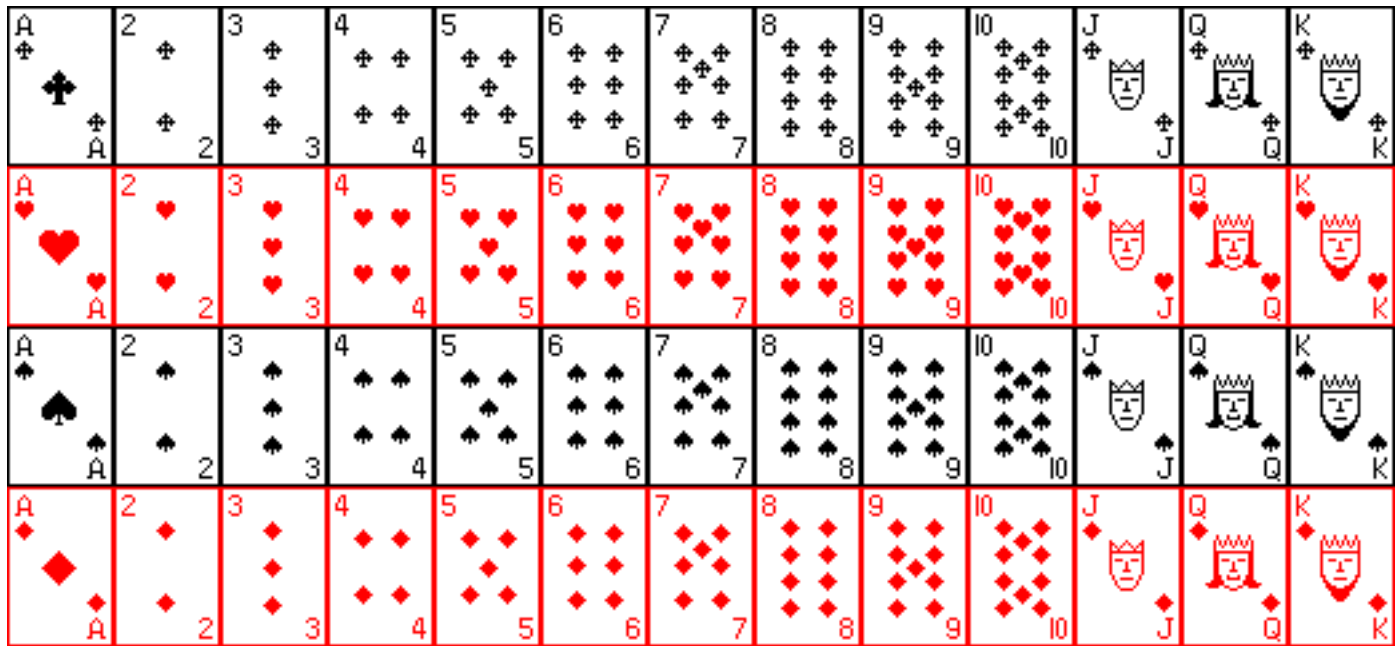
There are a few useful variations of the `drawImage()` command. For example, it is possible to scale the image as it is drawn to a specified width and height. This is done with the command

```
g.drawImage(img, x, y, width, height, this);
```

The parameters `width` and `height` give the size of the rectangle in which the image is displayed. Another version makes it possible to draw just part of the image. In the command:

```
g.drawImage(img, dest_x1, dest_y1, dest_x2, dest_y2,
            source_x1, source_y1, source_x2, source_y2, this);
```

the integers `source_x1`, `source_y1`, `source_x2`, and `source_y2` specify the top-left and bottom-right corners of a rectangular region in the source image. The integers `dest_x1`, `dest_y1`, `dest_x2`, and `dest_y2` specify the corners of a region in the destination graphics context. The specified rectangle in the image is drawn, with scaling if necessary, to the specified rectangle in the graphics context. For an example in which this is useful, consider a card game that needs to display 52 different cards. Dealing with 52 image files can be cumbersome and inefficient, especially for downloading over the Internet. So, all the cards might be put into a single image:



Now, only one Image object is needed. Drawing one card means drawing a rectangular region from the image. This technique is used in the following version of the [HighLow](#) card game from [Section 6.6](#):

(Applet "HighLowGUI2" would be displayed here
if Java were available.)

In this applet, the cards are drawn by the following method. The variable, `cardImages`, is a variable of type `Image` that represents the image of 52 cards that is shown above. Each card is 40 by 60 pixels. These numbers are used, together with the suit and value of the card, to compute the corners of the source and destination rectangles for the `drawImage()` command:

```
void drawCard(Graphics g, Card card, int x, int y) {
    // Draws a card as a 40 by 60 rectangle with
    // upper left corner at (x,y). The card is drawn
    // in the graphics context g. If card is null, then
    // a face-down card is drawn. The cards are taken
    // from an Image object that loads the image from
    // the file smallcards.gif.
    if (card == null) {
        // Draw a face-down card
        g.setColor(Color.blue);
        g.fillRect(x,y,40,60);
        g.setColor(Color.white);
        g.drawRect(x+3,y+3,33,53);
        g.drawRect(x+4,y+4,31,51);
    }
    else {
        int row = 0; // Which of the four rows contains this card?
        switch (card.getSuit()) {
            case Card.CLUBS:    row = 0; break;
            case Card.HEARTS:   row = 1; break;
            case Card.SPADES:   row = 2; break;
            case Card.DIAMONDS: row = 3; break;
        }
        int sx, sy; // Coords of upper left corner in the source image.
```



```

        sx = 40*(card.getValue() - 1);
        sy = 60*row;
        g.drawImage(cardImages, x, y, x+40, y+60,
                    sx, sy, sx+40, sy+60, this);
    }
} // end drawCard()

```

The variable `cardImages` is defined as an instance variable in the applet, and the image object is created in the `init()` method of the applet with the command:

```
cardImages = getImage( getCodeBase(), "smallcards.gif" );
```

The complete source code for this applet can be found in [HighLowGUI2.java](#).

Off-screen Images and Double Buffering

In addition to images in image files, objects of type `Image` can be used to represent images stored in the computer's memory. What makes such images particularly useful is that it is possible to draw to an `Image` in the computer's memory. This drawing is not visible to the user. Later, however, the image can be copied very quickly to the screen. In fact, this technique is used automatically in Swing to draw the components that you see on the screen. When the on-screen picture needs to be redrawn, the new picture is drawn step-by-step to an off-screen image. This can take some time. If all this drawing were done on screen, the user would see the image flicker as it is drawn. Instead, a complete new image replaces the old one on the screen almost instantaneously. The user doesn't see all the steps involved in redrawing. This technique makes smooth, flicker-free animation and dragging easy in Swing. (It is not at all easy or automatic when using the older AWT GUI components. This is one big advantage of Swing.)

The technique of drawing an off-screen image and then quickly copying the image to the screen is called **double buffering**. The name comes from the term "frame buffer," which refers to the region in memory that holds the image on the screen. (In fact, true double buffering uses two frame buffers. The video card can display either frame buffer on the screen and can switch instantaneously from one frame buffer to the other. One frame buffer is used to draw a new image for the screen. Then the video card is told to switch from one frame buffer to the other. No copying of memory is involved. Double-buffering as it is implemented in Java does require copying, which takes some time and is not perfectly flicker-free.)

It's possible to turn off double buffering in Swing (although there is little reason to do so). To help you understand the effect of double buffering, here are two applets that are identical, except that one uses double buffering and one does not. You can drag the red squares around the applets. I've added a lot of lines in the background to increase the time it takes to redraw the applet. You should notice an annoying flicker in the non-double-buffered applet on the left:

(Applets "NonDoubleBufferedDrag" and "DoubleBufferedDrag"
would be displayed here if Java were available.)

Swing's double buffering uses an off-screen image. Sometimes, it's useful to create your own off-screen images for other purposes. An off-screen `Image` object can be created by calling the instance method `createImage()`. This method is defined in the `Component` class, and so can be used just about anywhere in an applet's source code. The `createImage()` method takes two parameters to specify the width and height of the image to be created. For example,

```
Image offScreenImage = createImage(width, height);
```

Drawing to an off-screen image is done in the same way as any other drawing in Java, by using a graphics context. The `Image` class defines an instance method `getGraphics()` that returns a `Graphics` object that can be used for drawing on the off-screen image. (This works only for off-screen images. If you try to do this with an `Image` from a file, an error will occur.) That is, if `offScreenImage` is a variable of type

Image that refers to an off-screen image, you can say

```
Graphics offscreenGraphics = offScreenImage.getGraphics();
```

Then, any drawing operations performed with the graphics context `offscreenGraphics` are applied to the off-screen image. For example, `"offscreenGraphics.drawRect(10,10,50,100);"` will draw a 50-by-100-pixel rectangle on the off-screen image. Once a picture has been drawn on the off-screen image, the picture can be copied into another graphics context, using the graphics context's `drawImage()` method. For example: `g.drawImage(offScreenImage, 0, 0, null)`. For an off-screen image, the file parameter to `drawImage()` can be null. (Since the image is already in memory, there is no need for an "ImageObserver" to wait for the image to be loaded from a file.)

Off-screen images can be used to solve one problem that we have seen in many of our sample applets. In many cases, we have had no convenient way of remembering what was drawn on an applet, so that we were unable restore the drawing when necessary. For example, in the [paint applet](#) in [Section 6.6](#), the user's sketch will disappear if the applet is covered up and then uncovered. An off-screen image can be used to solve this problem. The idea is simple: Keep a copy of the drawing in an off-screen image. When the component needs to be redrawn, copy the off-screen image onto the screen. This method is used in the improved paint program at the end of this section.

When used in this way, the off-screen image should always contain a copy of the picture on the screen. The `paintComponent()` method copies this off-screen image to the screen. This will refresh the picture when it is covered and uncovered. The actual drawing of the picture should take place elsewhere. (Occasionally, it makes sense to draw some extra stuff on the screen, on top of the image from the off-screen image. For example, a hilite or a shape that is being dragged might be treated in this way. These things are not permanently part of the image. The permanent image is safe in the off-screen image, and it can be used to restore the on-screen image when the hilite is removed or the shape is dragged to a different location. We will use this technique in the next example.)

There are two approaches to keeping the image on the screen synchronized with the image in the off-screen image. In the first approach, in order to change the image, you make the change to the off-screen image and then call `repaint()` to copy the modified image to the screen. This is safe and easy, but not always efficient. The second approach is to make every change twice, once to the off-screen image and once to the screen. This keeps the two images the same, but it requires some care to make sure that exactly the same drawing is done in both (and it violates the rule about doing drawing operations only inside `paintComponent()` methods).

When using an off-screen image as a backup for the picture displayed on a component, the size of the off-screen image should be the same as the size of the component. This raises the problem of where in the program the image should be created. If the off-screen image is to fill an entire applet, then the image can be created in the applet's `init()` method with the command:

```
offScreenImage = createImage(getSize().width,getSize().height);
```

However, components other than applets do not have convenient `init()` methods for initialization. They have constructors, but the size of a component is not known when its constructor is executed, so the above command will not work in a constructor. An alternative is to create the off-screen image on demand, when it is needed. We can even allow for changes in size of a component if we make a new off-screen image whenever the size changes. Here is some sample code that implements this idea. A method named `checkOffScreenImage()` will create the off-screen image when necessary. This method should always be called before using the off-screen image. For example, it is called in the `paintComponent()` method before copying the image to the screen.

```
/* Some variables used for double-buffering. */
```

```
Image OSI; // The off-screen image (created in paintComponent()).
```

```

int widthOfOSI, heightOfOSI; // Current width and height of OSI.
                                // These are checked against the size
                                // of the component, to detect any change
                                // in the component's size.  If the size
                                // has changed, a new OSI is created.
                                // The picture in the off-screen image
                                // is lost when that happens.

void checkOffScreenImage() {
    // This method will create the off-screen image if it has not
    // already been created or if the component's size has changed.
    // It should always be called before using the off-screen
    // image in any way.
    if (OSI == null || widthOfOSI != getSize().width
        || heightOfOSI != getSize().height) {
        // OSI doesn't yet exist, or else it exists but has a
        // different size from the component's current size.
        // Create a new OSI, and fill it with the component's
        // background color.
        OSI = null; // If OSI already exists, this frees up the memory.
        widthOfOSI = getSize().width;
        heightOfOSI = getSize().height;
        OSI = createImage(widthOfOSI, heightOfOSI);
        Graphics OSGr = OSC.getGraphics();
        OSGr.setColor(getBackground());
        OSGr.fillRect(0, 0, widthOfOSI, heightOfOSI);
        OSGr.dispose(); // Free operating system resources.
    }
}

public void paintComponent(Graphics g) {
    // Paint the component by copying the off-screen image onto
    // the screen.  First, call checkOffScreenImage() to make
    // sure that the off-screen image is ready.
    // (Note that since the image fills the entire component,
    // it is not necessary to call super.paintComponent(g).)

    checkOffScreenImage();
    g.drawImage(OSI, 0, 0, null); // Copy OSI onto the screen.

    // Note: At this point, we could draw hiliting or other extra
    // stuff on top of the picture in the off-screen image.
}

```

Note that the contents of the off-screen image are lost if the size changes. If this is a problem, you can consider copying the contents of the old off-screen image to the new one before discarding the old image. You can do this with `drawImage()`, and you can even scale the image to fit the new size if you want. However, the results of scaling are not always attractive.

Here is an applet that demonstrates some of these ideas. Draw red lines by clicking and dragging on the applet. Draw blue rectangles by right-clicking and dragging. Hold down the shift key and click to clear the applet. Notice that as you drag the mouse, the figure that you are drawing stretches between the current mouse position and the point where you started dragging. This effect is sometimes called a **rubber band cursor**:

(Applet "RubberBand" would be displayed here
if Java were available.)

In this applet, a copy of the picture that you've drawn is kept in an off-screen image. If you cover the applet and uncover it, the picture is restored by copying this backup image onto the screen. When you drag the mouse, the figure that you are drawing is not added to the off-screen image. The `paintComponent()` method simply draws the new figure on top of the backup image. The backup image is not changed, and as you move the mouse around, you can see that it is still there, "underneath" the figure you are sketching. The new figure is only added to the off-screen image when you release the mouse button. To see how all this works in detail, check the source code, [RubberBand.java](#).

There is one other point of interest in the above applet. To draw a rectangle in Java, you need to know the coordinates of the upper left corner, the width, and the height. However, when a rectangle is drawn in this applet, the available data consists of two corners of the rectangle: the starting position of the mouse and its current position. From these two corners, the left edge, the top edge, the width, and the height of the rectangle have to be computed. This can be done as follows:

```
void drawRectUsingCorners(Graphics g, int x1, int y1, int x2, int y2) {
    // Draw a rectangle with corners at (x1,y1) and (x2,y2).
    int x,y; // Coordinates of the top-left corner.
    int w,h; // Width and height of rectangle.
    if (x1 < x2) { // x1 is the left edge
        x = x1;
        w = x2 - x1;
    }
    else { // x2 is the left edge
        x = x2;
        w = x1 - x2;
    }
    if (y1 < y2) { // y1 is the top edge
        y = y1;
        h = y2 - y1;
    }
    else { // y2 is the top edge
        y = y2;
        h = y1 - y2;
    }
    g.drawRect(x, y, w, h); // Draw the rect.
}
```

Rectangles, Clipping, and Repainting

The example we've just looked at has one glaring inefficiency: Every time the user drags the mouse, the entire applet is repainted, even though only a small part of the picture might need to be changed. It's possible to improve on this by repainting only a part of the applet. There is a second version of the `repaint()` command that makes this possible. If `comp` is a variable that refers to some component, then

```
comp.repaint( x, y, width, height );
```

tells the system that a rectangular area in the component needs to be repainted. The first two parameters, `x` and `y`, specify the upper left corner of the rectangle and the next two parameters give the width and height of the rectangle. In response to this, the system will call `paintComponent()` as usual, but the graphics context will be set up for drawing only in the specified region. This is done by setting the **clip region** of the graphics context. The clip region of a graphics context specifies the area where drawing can occur. Any

attempt to use the graphics context to draw outside the clip region is ignored. (If part of a shape lies outside the clip region, that part is "clipped off" before the shape is drawn on the screen.) Only the pixels inside the clip region need to have their color set, and this can be much more efficient than setting the color of every pixel in the component. When an off-screen image is copied onto the component, only the part that lies within the clip region is actually copied.

The techniques covered in this section can be used to improve the simple painting program from [Section 6.6](#). The new version uses an off-screen image to save a copy of the user's work, and it uses the version of `repaint()` discussed above. As before, the user can draw a free-hand sketch. However, in this version, the user can also choose to draw several shapes by selecting from the pop-up menu in the upper right. Try it out! Check that when you cover up the applet with another window, your drawing is still there when you uncover it.

(Applet "SimplePaint3" would be displayed here
if Java were available.)

The source code for this improved paint applet is in the file [SimplePaint3.java](#). It uses an off-screen image pretty much in the way described above. The `paintComponent()` method copies the off-screen image to the screen, and as the user drags the mouse, clipping is used to restrict the drawing to the region that actually needs to be changed.

In this applet, curves are handled differently from the other shapes. Suppose that the user is sketching a curve and that the user moves the mouse from the point `(prevX,prevY)` to the point `(mouseX,mouseY)`. The applet responds to this by drawing a line segment *in the off-screen image* from `(prevX,prevY)` to `(mouseX,mouseY)`. **To make this change appear on the screen, a rectangle that contains these two points must be copied from the off-screen image onto the screen. This is accomplished in the applet by calling `repaint(x,y,w,h)` with appropriate values for the parameters.**

When the user is sketching one of the other shapes in the applet, the rubber band cursor technique is used. That is, while the user is dragging the mouse, the shape is drawn by the `paintComponent()` method on top of the picture from the off-screen image. Let's say, for example, that the user is drawing a rectangle. Suppose that the user starts by pressing the mouse at the point `(startX,startY)`. Consider what happens later, when the user drags the mouse from the point `(prevX,prevY)` to the point `(mouseX,mouseY)`. At the beginning of this motion, a rectangle is shown on the screen with corners at `(startX,startY)` and `(prevX,prevY)`. In response to the motion, this rectangle must be removed and a new one with corners at `(startX,startY)` and `(mouseX,mouseY)` should appear. This can be accomplished by changing the values of the variables that tell `paintComponent()` where to draw the rectangle and by calling `repaint(x,y,w,h)` twice: once to repaint the area occupied by the old rectangle and once to repaint the area that will be occupied by the new rectangle. (The system will actually combine the two operations into a single call to `paintComponent()`.)

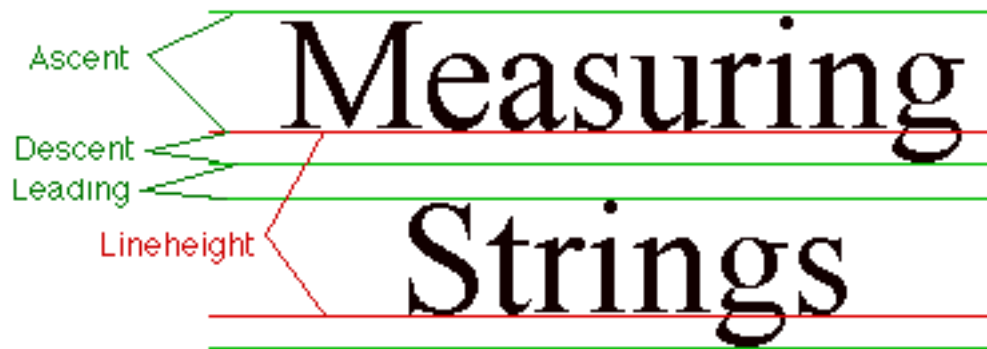
This version of "SimplePaint" is not really all that simple. There are a lot of details to take care of. I urge you to look at the [source code](#) to see how it's done.

FontMetrics

In the rest of this section, we turn from `Images` to look briefly at another aspect of Java graphics.

Often, when drawing a string, it's important to know how big the image of the string will be. You need this information if you want to center a string on an applet. Or if you want to know how much space to leave between two lines of text, when you draw them one above the other. Or if the user is typing the string and you want to position a cursor at the end of the string. In Java, questions about the size of a string are answered by an object belonging to the standard class `java.awt.FontMetrics`.

There are several lengths associated with any given font. Some of them are shown in this illustration:



The red lines in the illustration are the **baselines** of the two lines of text. The suggested distance between two baselines, for single-spaced text, is known as the **lineheight** of the font. The **ascent** is the distance that tall characters can rise above the baselines, and the **descent** is the distance that tails like the one on the letter g can descend below the baseline. The ascent and descent do not add up to the lineheight, because there should be some extra space between the tops of characters in one line and the tails of characters on the line above. The extra space is called **leading**. All these quantities can be determined by calling instance methods in a `FontMetrics` object. There are also methods for determining the width of a character and the width of a string.

If `F` is a font and `g` is a graphics context, you can get a `FontMetrics` object for the font `F` by calling `g.getFontMetrics(F)`. If `fm` is a variable that refers to the `FontMetrics` object, then the ascent, descent, leading, and lineheight of the font can be obtained by calling `fm.getAscent()`, `fm.getDescent()`, `fm.getLeading()`, and `fm.getHeight()`. If `ch` is a character, then `fm.charWidth(ch)` is the width of the character when it is drawn in that font. If `str` is a string, then `fm.stringWidth(str)` is the width of the string. For example, here is a `paintComponent()` method that shows the message "Hello World" in the exact center of the component:

```
public void paintComponent(Graphics g) {
    int width, height; // Width and height of the string.
    int x, y;          // Starting point of baseline of string.
    Font F = g.getFont(); // What font will g draw in?
    FontMetrics fm = g.getFontMetrics(F);
    width = fm.stringWidth("Hello World");
    height = fm.getAscent(); // Note: There are no tails on
                           // any of the chars in the string!
    x = getSize().width / 2 - width / 2; // Go to center and back up
                                         // half the width of the
                                         // string.
    y = getSize().height / 2 + height / 2; // Go to center, then move
                                         // down half the height of
                                         // the string.

    g.drawString("Hello World", x, y);
}
```


Section 7.2

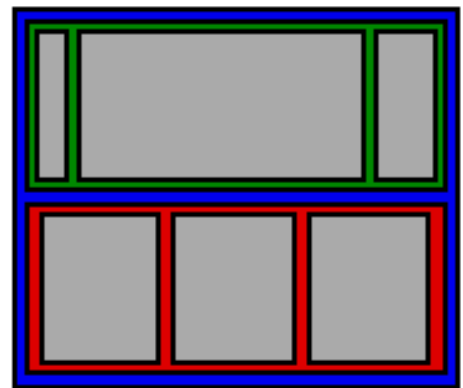
More about Layouts and Components

SWING INCLUDES A VARIETY of GUI components. We have already encountered a few of these, such as `JApplet`, `JButton`, and `JPanel`. In the next few sections, we will be studying Swing components in more detail.

Most Swing components are defined by subclasses of the class `javax.swing.JComponent`. A `JComponent` cannot stand on its own. It must be contained in some other component. We have seen, for example, that `JPanel`s can act as containers for other `JComponents`. At the top level of this containment hierarchy are classes such as `JApplet`. A `JApplet` is not a `JComponent`, but it can serve as a container for `JComponents`. A `JApplet` is a top-level container that is meant to appear on a Web page. In [Section 7](#), we'll see two more top-level container classes, `JFrame` and `JDialog`, which can be used to create independent windows on the computer screen.

The basic properties of components and containers are actually defined by the AWT classes `java.awt.Component` and `java.awt.Container`. Occasionally, you will see these classes used in Swing. For example, the `getContentPane()` method in a `JApplet` has a return type of `Container` rather than `JPanel` or `JComponent` as you might expect.

A `JPanel` is a container that is itself a `JComponent`. A `JPanel` can contain other components, and it can in turn be contained in another component. The fact that panels can contain other panels means that you can have many levels of components containing other components, as shown in the illustration on the right. Several other classes, such as `Box` and `TabbedPane`, also define components that can be used as containers. This leads to two questions: How are components added to a container? How are their sizes and positions controlled?



Three Panels, shown in color, containing six additional Components, shown in gray.

The sizes and positions of the components in a container are usually controlled by a **layout manager**. Different layout managers implement different ways of arranging components. There are several predefined layout manager classes, including `FlowLayout`, `GridLayout`, `BorderLayout`, `BoxLayout`, `CardLayout` and `GridBagLayout`. All these classes are defined in the package `java.awt`. It is also possible to define new layout managers, if none of these suit your purpose. Every container is assigned a default layout manager when it is first created. For `JPanel`s, the default layout manager belongs to the class `FlowLayout`. The content pane of a `JApplet` uses a `BorderLayout` by default. You can change the layout manager of a container using its `setLayout()` method.

It is even possible to set the `LayoutManager` of a container to be `null`. This allows you to take complete charge of laying out the components in the container. I will discuss this possibility and give an example in the last part of [Section 4](#).

As for adding components to a container, that's easy. You just use one of the container's `add()` methods. There are several `add()` methods. Which one you should use depends on what type of `LayoutManager` is being used by the container, so I will discuss the appropriate `add()` methods as I go along.

I have often found it to be fairly difficult to get the exact layout that I want in my applets and windows. I will briefly discuss several layout manager classes here, but using them well will require practice and experimentation.

FlowLayout

A `FlowLayout` simply lines up its components without trying to be particularly neat about it. After laying out as many items as will fit in a row across the container, it will move on to the next row. The components in a given row can be either left-aligned, right-aligned, or centered, and there can be horizontal and vertical gaps between components. If the default constructor, `new FlowLayout()` is used, then the components on each row will be

centered and the horizontal and vertical gaps will be five pixels. The default layout for a `JPanel` uses gaps of this size. The constructor

```
FlowLayout(int align, int hgap, int vgap)
```

can be used to specify alternative alignment and gaps. The possible values of `align` are `FlowLayout.LEFT`, `FlowLayout.RIGHT`, and `FlowLayout.CENTER`. A nifty trick is to use a very large value of `hgap`. This forces the `FlowLayout` to put exactly one component in each row, since there won't be room on a single row for two components and the horizontal gap between them. The appropriate `add()` method for `FlowLayouts` has a single parameter of type `Component`, specifying the component to be added.

For example, suppose that we wanted an applet to contain one button, located in the upper right corner of the applet. The default layout manager for an applet's content pane is a `BorderLayout`. We need to give the content pane a `FlowLayout` with right alignment. This will shove the button to the right edge of the applet. The following `init()` method will do this:

```
public void init() {
    getContentPane().setLayout( new FlowLayout(FlowLayout.RIGHT, 5, 5) );
    getConetntPane().add( new JButton("Press me!") );
}
```

Note again that it is the applet's content pane that actually holds components, and it is the content pane that needs a layout manager. It is an error to try to set a layout manager for a `JApplet` itself.

BoxLayout and the Box Class

A `BoxLayout` simply lines up components in a single horizontal row or in a single vertical column. `BoxLayouts` are generally used with objects belonging to the class `javax.swing.Box`. A `Box` is just a container that uses a `BoxLayout`. The `Box` class contains two static methods for creating boxes:

```
Box.createHorizontalBox();
and
Box.createVerticalBox();
```

These methods are used instead of a constructor to create box objects. For example, if you want a `Box` to contain a horizontal row of components, you can create it with the command:

```
Box hbox = Box.createHorizontalBox();
```

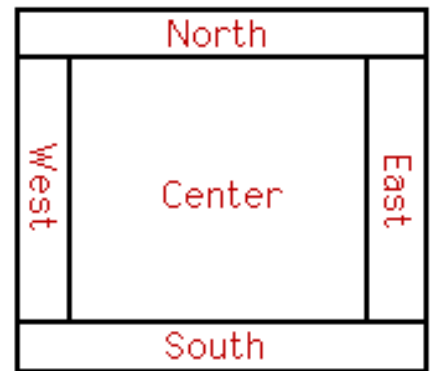
Components are added to a box using an `add()` method with one parameter, which specifies the component that is to be added. The `Box` class has several static methods that can be used to create specialized components for adding space to a box layout. For example, if `width` is an integer, then `Box.createHorizontalStrut(width)` creates a component that is invisible except that it has the specified width and so takes up that amount of horizontal space. You can add a horizontal strut between two components in a horizontal box layout to leave space between the components. Similarly, `Box.createVerticalStrut(height)` creates an invisible component that has the specified height. For example, the following commands create a `Box` that contains four (useless) buttons in a horizontal row, with ten pixels of space between the second and third button:

```
Box hbox = Box.createHorizontalBox();
hbox.add( new JButton("First") );
hbox.add( new JButton("Second") );
hbox.add( Box.createHorizontalStrut(10) );
hbox.add( new JButton("Third") );
hbox.add( new JButton("Fourth") );
```

Horizontal Boxes can be used for the "toolbars" that you see in many graphical user interfaces.

BorderLayout

A `BorderLayout` places one component in the center of a container. This central component is surrounded by up to four other components that border it to the "North", "South", "East", and "West", as shown in the diagram at the right. Each of the four bordering components is optional. The layout manager first allocates space to the bordering components. Any space that is left over goes to the center component.



If a container uses a `BorderLayout`, then components should be added to the container using a version of the `add()` method that has two parameters. The first parameter is the component that is being added to the container. The second parameter specifies where the component is to be placed. It must be one of the constants `BorderLayout.CENTER`, `BorderLayout.NORTH`, `BorderLayout.SOUTH`, `BorderLayout.EAST`, or `BorderLayout.WEST`. If the second parameter is omitted, then `BorderLayout.CENTER` is used by default. For example, the following code creates a panel with `drawArea` as its center component and with scroll bars to the right and below:

```
JPanel panel = new JPanel();
panel.setLayout(new BorderLayout());
    // To use BorderLayout with a JPanel, you have
    // to change the panel's layout manager; otherwise,
    // a FlowLayout is used. Alternatively, you
    // can provide the layout manager as a
    // parameter to the constructor:
    // panel = new JPanel( new BorderLayout() );
panel.add(drawArea, BorderLayout.CENTER);
    // Assume drawArea already exists.
panel.add(hScroll, BorderLayout.SOUTH);
    // Assume hScroll is a horizontal scroll bar
    // component that already exists.
panel.add(vScroll, BorderLayout.EAST);
    // Assume vScroll is a vertical scroll bar
    // component that already exists.
```

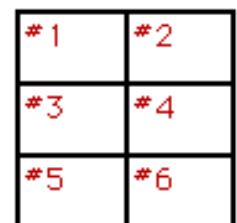
Sometimes, you want to leave space between the components in a container. You can specify horizontal and vertical gaps in the constructor of a `BorderLayout` object. For example, if you say

```
panel.setLayout(new BorderLayout(5,7));
```

then the layout manager will insert horizontal gaps of 5 pixels between components and vertical gaps of 7 pixels between components. The horizontal gap is inserted between the center and west components and between the center and east components; the vertical gap is inserted between the center and north components and between the center and south components. (The default layout for a JApplet's content pane is a `BorderLayout` with no horizontal or vertical gap.)

GridLayout

A `GridLayout` lays out components in a grid of equal sized rectangles. The illustration shows how the components would be arranged in a grid layout with 3 rows and 2 columns. If a container uses a `GridLayout`, the appropriate `add` method takes a single parameter of type `Component` (for example: `add(myButton)`). Components are added to the grid in the order shown; that is, each row is filled from left to right before going on the next row.



The constructor for a `GridLayout` with `R` rows and `C` columns takes the form "`new GridLayout(R,C)`". If you want to leave horizontal gaps of `H` pixels between columns and vertical gaps of `V` pixels between rows, use "`new GridLayout(R,C,H,V)`" instead.

When you use a `GridLayout`, it's probably good form to add just enough components to fill the grid. However, this

is not required. In fact, as long as you specify a non-zero value for the number of rows, then the number of columns is essentially ignored. The system will use just as many columns as are necessary to hold all the components that you add to the container. If you want to depend on this behavior, you should probably specify zero as the number of columns. You can also specify the number of rows as zero. In that case, you must give a non-zero number of columns. The system will use the specified number of columns, with just as many rows as necessary to hold the components that are added to the container.

Horizontal grids, with a single row, and vertical grids, with a single column, are very common. For example, suppose that `button1`, `button2`, and `button3` are buttons and that you'd like to display them in a horizontal row in a panel. If you use a horizontal grid for the panel, then the buttons will completely fill that panel and will all be the same size. The panel can be created as follows:

```
JPanel buttonBar = new JPanel();
buttonBar.setLayout(new GridLayout(1,3));
// (Note: The "3" here is pretty much ignored, and
// you could also say "new GridLayout(1,0)".
// To leave gaps between the buttons, you could use
// "new GridLayout(1,0,5,5)".)
buttonBar.add(button1);
buttonBar.add(button2);
buttonBar.add(button3);
```

You might find this button bar to be more attractive than the ones in the examples in the [Section 6.6](#), which used the default `FlowLayout` layout manager.

GridBagLayout

A `GridBagLayout` is similar to a `GridLayout` in that the container is broken down into rows and columns of rectangles. However, a `GridBagLayout` is much more sophisticated because the rows do not all have to be of the same height, the columns do not all have to be of the same width, and a component in the container can spread over several rows and several columns. There is a separate class, `GridBagConstraints`, that is used to specify the position of a component, the number of rows and columns that it occupies, and several additional properties of the component.

Using a `GridBagLayout` is rather complicated, and I have used it on exactly two occasions in my own Java programming career. I will not explain it here; if you are interested, you should consult a Java reference.

CardLayout

`CardLayouts` differ from other layout managers in that in a container that uses a `CardLayout`, only one of its components is visible at any given time. Think of the components as a set of "cards". Only one card is visible at a time, but you can flip from one card to another. Methods are provided in the `CardLayout` class for flipping to the first card, to the last card, and to the next card in the deck. A name can be specified for each card as it is added to the container, and there is a method in the `CardLayout` class for flipping directly to the card with a specified name.

Suppose, for example, that you want to create a `JPanel` that can show any one of three `JPanels`: `panel1`, `panel2`, and `panel3`. Assume that `panel1`, `panel2`, and `panel3` have already been created:

```
cardPanel = new JPanel();
// assume cardPanel is declared as an instance variable
// so that it can be used in other methods
cards = new CardLayout();
// assume cards is declared as an instance variable
// so that it can be used in other methods
cardPanel.setLayout(cards);
cardPanel.add(panel1, "First");
// add panel1 with name "First"
cardPanel.add(panel2, "Second");
```

```
// add panel2 with name "Second"
cardPanel.add(panel3, "Third");
// add panel3 with name "Third"
```

Elsewhere in your program, you could show `panel1` by saying

```
cards.show(cardPanel, "First");
```

or

```
cards.first(cardPanel);
```

Other methods that are available are `cards.last(cardPanel)`, `cards.next(cardPanel)`, and `cards.previous(cardPanel)`. Note that each of these methods takes the container as a parameter. To use a `CardLayout` effectively, you'll need to have instance variables to record both the layout manager (`cards` in the example) and the container (`cardPanel` in the example). You need both of these objects in order to flip from one card to another.

An Example

To finish this survey of layout managers, here is an applet that demonstrates layout managers of various types:

(Applet "LayoutDemo" would be displayed here
if Java were available.)

The applet itself uses a `BorderLayout` with vertical gaps of 3 pixels. These gaps show up in blue. The Center component of the applet is a `JPanel`, which uses a `CardLayout` as its layout manager. The layout contains eight cards. Each card is itself another panel that contains several buttons. Each card uses a different type of layout manager (several of which are extremely stupid choices for laying out buttons).

The North component of the applet is a `JComboBox`, which contains the names of the eight panels in the card layout. The user can switch among the cards by selecting items from this menu. The South component of the applet is a `JLabel` that displays an appropriate message whenever the user clicks on a button or chooses an item from the `JComboBox`.

The source code for this applet is in the file [LayoutDemo.java](#). It consists mainly of a long `init()` method that creates all the buttons, panels, and other components and lays out the applet.

Borders and Insets

Swing makes it very easy to add decorative borders around the edges of a `JComponent`. The class `javax.swing.BorderFactory` contains a large number of static methods for creating borders. For example, the function

```
BorderFactory.createLineBorder(Color.black)
```

returns an object that represents a one-pixel wide black line around the outside of a component. If `comp` is a `JComponent`, a border can be added to `comp` using its `setBorder()` method. For example:

```
comp.setBorder( BorderFactory.createLineBorder(Color.black) );
```

When a border has been set for a `JComponent`, the border is drawn automatically, without any further effort on the part of the programmer. The border is drawn along the edges of the component, just inside its boundary. The layout manager of a `JPanel` or other container will take the space occupied by the border into account. The components that are added to the container will be displayed in the area inside the border. I don't recommend using a border on a `JPanel` that is being used as a drawing surface. However, if you do this, you should take the border into account. If you draw in the area occupied by the border, that part of your drawing will be covered by the border.

Here are some of the static methods that can be used to create borders:

- `BorderFactory.createEmptyBorder(top, left, bottom, right)` -- leaves an empty border

around the edges of a component. Nothing is drawn in this space, so the background color will appear in the area occupied by the border. The parameters are integers that give the width of the border along the top, left, bottom, and right edges of the component. This is actually very useful when used on a `JPanel` that contains other components. It puts some space between the components and the edge of the panel.

- `BorderFactory.createLineBorder(color, thickness)` -- draws a line around all four edges of a component. The first parameter is of type `Color` and specifies the color of the line. The second parameter is an integer that specifies the thickness of the border. If the second parameter is omitted, a line of thickness 1 is drawn.
- `BorderFactory.createMatteBorder(top, left, bottom, right, color)` -- is similar to `createLineBorder`, except that you can specify individual thicknesses for the top, left, bottom, and right edges of the component.
- `BorderFactory.createEtchedBorder()` -- creates a border that looks like a groove etched around the boundary of the component. The effect is achieved using lighter and darker shades of the component's background color, and it does not work well with every background color.
- `BorderFactory.createLoweredBevelBorder()` -- gives a component a three-dimensional effect that makes it look like it is lowered into the computer screen. As with an `EtchedBorder`, this only works well for certain background colors.
- `BorderFactory.createRaisedBevelBorder()` -- similar to a `LoweredBevelBorder`, but the component looks like it is raised above the computer screen.
- `BorderFactory.createTitledBorder(title)` -- creates a border with a title. The title is a `String`, which is displayed in the upper left corner of the border.

There are many other methods in the `BorderFactory` class, most of them providing variations of the basic border styles given here. The following applet shows six components with six different border styles. The text in each component is the command that created the border for that component:

(Applet "BorderDemo" would be displayed here
if Java were available.)

Since a `JApplet` is not a `JComponent`, it's not possible to set a `Border` object for a `JApplet`. There is, however, another way to add a border of color around the edges. An applet can use "insets" to leave space around the edges of the applet where the background color of the applet will show through. To do this, define the method `public Insets getInsets()` in your subclass of `JApplet`. This method should return an object of type `Insets`, which specifies the width of the border along the top, left, bottom, and right edges of the applet. The system will call your method to determine how much space to leave. For example, if your subclass of `JApplet` includes the method definition:

```
public Insets getInsets() {
    return new Insets(5,5,5,5);
}
```

then there will be a 5-pixel-wide border around the edges of the applet where the background color of the applet will show. To specify the color, you can set the applet's background color in its `init()` method. Note that `Insets` should not be used with `JComponents`. For a `JComponent`, you can use `BorderFactory.createEmptyBorder()` to accomplish the same thing.

The `LayoutDemo` applet uses `Insets` to leave a 3-pixel border around the outside of the applet, where the blue background color of the applet shows through. This is different from the 3-pixel blue gap between the components in the applet's content pane, where the blue gap is a feature of the content pane's `BorderLayout`. It's the background color of the content pane, not of the applet, that shows through the spaces in the `BorderLayout`. To set up the colors, the `init()` method of the applet sets the background color for **both the applet and for its content pane to blue**. Since the default layout used for a content pane has no vertical gap, the `init()` method also installs a different layout manager for the content pane. All this is done with the following commands:

```
setBackground(Color.blue);
getContentPane().setBackground(Color.blue);
getContentPane().setLayout(new BorderLayout(3,3));
```

Section 7.3

Basic Components and Their Events

THIS SECTION DISCUSSES some of the GUI interface elements that are represented by subclasses of `JComponent`. The treatment here is brief and covers only the basic uses of each component type. After you become familiar with the basics, you might want to consult a Java reference for more details. I will give some examples of programming with components in the [next section](#).

The exact appearance of a Swing component and the way that the user interacts with the component are not fixed. They depend on the **look-and-feel** of the user interface. While Swing supports a default look-and-feel, which is probably the one that you will see most often, it is possible to change the look-and-feel. For example, a Windows look-and-feel could be used to make a Java program that is running on a Windows computer look more like a standard Windows program. While this can improve the user's experience, it means that some of the details that I discuss will have to be qualified with the phrase "depending on the look-and-feel."

The `JComponent` class itself defines many useful methods that can be used with components of any type. We've already used some of these in examples. Let `comp` be a variable that refers to any `JComponent`. Then the following methods are available (among many others):

- `comp.getSize()` is a function that returns an object belonging to the class `Dimension`. This object contains two instance variables, `comp.getSize().width` and `comp.getSize().height`, that give the current size of the component. You can also get the height and width more directly by calling `comp.getHeight()` and `comp.getWidth()`. One warning: When a component is first created, its size is zero. The size will be set later, probably by a layout manager. A common mistake is to check the size of a component before that size has been set, such as in a constructor.
- `comp.setEnabled(true)` and `comp.setEnabled(false)` can be used to enable and disable the component. When a component is disabled, its appearance changes, and the user cannot do anything with it. There is a boolean-valued function, `comp.isEnabled()` that you can call to discover whether the component is enabled.
- `comp.setVisible(true)` and `comp.setVisible(false)` can be called to hide or show the component.
- `comp.setBackground(color)` and `comp.setForeground(color)` set the background and foreground colors for the component. If no colors are set for a component, the colors are determined by the look-and-feel.
- `comp.setOpaque(true)` tells the component that the area occupied by the component should be filled with the component's background color before the content of the component is painted. In the default look-and-feel, only `JLabels` are non-opaque. A non-opaque, or "transparent", component ignores its background color and simply paints its content over the content of its container. This usually means that it inherits the background color from its container.
- `comp.setFont(font)` sets the font that is used for text displayed on the component. The parameter is an object of type `java.awt.Font`.
- `comp.setToolTipText(string)` sets the specified string as a "tool tip" for the component. The tool tip is displayed if mouse cursor is in the component and the mouse is not moved for a few seconds. The tool tip should give some information about the meaning of the component or how to use it.
- `comp.setCursor(cursor)` sets the cursor image that represents the mouse position when the mouse cursor is inside this component. The parameter is an object belonging to the class `java.awt.Cursor`. Generally, this parameter takes the form

`Cursor.getPredefinedCursor(id)` where `id` is one of several constants defined in the `Cursor` class. The most useful values are probably `Cursor.HAND_CURSOR`, `Cursor.CROSSHAIR_CURSOR`, `Cursor.WAIT_CURSOR`, and `Cursor.DEFAULT_CURSOR`. For example, if you would like the cursor to appear as a little pointing hand when the mouse is in the component `comp`, you can use:

```
comp.setCursor(Cursor.getPredefinedCursor(Cursor.HAND_CURSOR));
```

- **`comp.setPreferredSize(size)`** sets the size at which the component should be displayed, if possible. The parameter is of type `Dimension`, and a call to this method usually looks something like `"setPreferredSize(new Dimension(100, 50))"`. The preferred size is used as a hint by layout managers, but will not be respected in all cases. In a `BorderLayout`, for example, the preferred size of the Center component is irrelevant, but the preferred sizes of the North, South, East, and West components are used by the layout manager to decide how much space to allow for those components. Standard components generally compute a correct preferred size automatically, but it can be useful to set it in some cases. For example, if you use a `JPanel` as a drawing surface, it might be a good idea to set a preferred size for it.
- **`comp.getParent()`** is a function that returns a value of type `java.awt.Container`. The return value is the container that directly contains the component, if any. For a top-level component such as a `JApplet`, the value will be null.
- **`comp.getLocation()`** is a function that returns the location of the top-left corner of the component. The location is specified in the coordinate system of the component's parent. The returned value is an object of type `Point`. An object of type `Point` contains two instance variables, `x` and `y`.

For the rest of this section, we'll look at subclasses of `JComponent` that represent common GUI components. Remember that using any component is a multi-step process. The component object must be created with a constructor. It must be added to a container. In many cases, a listener must be registered to respond to events from the component. And in some cases, a reference to the component must be saved in an instance variable so that the component can be manipulated by the program after it has been created.

The JButton Class

An object of class `JButton` is a push button. You've already seen buttons used in the previous chapter, but we can use a review of `JButtons` as a reminder of what's involved in using components, events, and listeners. (Some of the methods described here are new.)

- **Constructors:** The `JButton` class has a constructor that takes a string as a parameter. This string becomes the text displayed on the button. For example: `stopGoButton = new JButton("Go")`
- **Events:** When the user clicks on a button, the button generates an event of type `ActionEvent`. This event is sent to any listener that has been registered with the button.
- **Listeners:** An object that wants to handle events generated by buttons must implement the `ActionListener` interface. This interface defines just one method, `"public void actionPerformed(ActionEvent evt)"`, which is called to notify the object of an action event.
- **Registration of Listeners:** In order to actually receive notification of an event from a button, an `ActionListener` must be registered with the button. This is done with the button's `addActionListener()` method. For example: `stopGoButton.addActionListener(buttonHandler);`
- **Event methods:** When `actionPerformed(evt)` is called by the button, the parameter, `evt`, contains information about the event. This information can be retrieved by calling methods in the `ActionEvent` class. In particular, `evt.getActionCommand()` returns a `String` giving the

command associated with the button. By default, this command is the text that is displayed on the button. The method `evt.getSource()` returns a reference to the Object that produced the event, that is, to the `JButton` that was pressed. The return value is of type `Object`, not `JButton`, because other types of components can also produce `ActionEvents`.

- **Component methods:** There are several useful methods in the `JButton` class. For example, `stopGoButton.setText("Stop")` changes the text displayed on the button to "Stop". And `stopGoButton.setActionCommand("sgb")` changes the action command associated to this button for action events.

Of course, `JButtons` also have all the general `Component` methods, such as `setEnabled()` and `setFont()`. The `setEnabled()` and `setText()` methods of a button are particularly useful for giving the user information about what is going on in the program. A disabled button is better than a button that gives an obnoxious error message such as "Sorry, you can't click on me now!"

By the way, it's possible for a `JButton` to display an **icon** instead of or in addition to the text that it displays. An icon is simply a small image. Several other components can also display icons. However, I will not cover this aspect of Swing in this book. Consult a Java reference if you are interested.

The JLabel Class

`JLabels` are certainly the simplest type of component. An object of type `JLabel` is just a single line of text. The text cannot be edited by the user, although it can be changed by your program. The constructor for a `JLabel` specifies the text to be displayed:

```
JLabel message = new JLabel("Hello World!");
```

There is another constructor that specifies where in the label the text is located, if there is extra space. The possible alignments are given by the constants `JLabel.LEFT`, `JLabel.CENTER`, and `JLabel.RIGHT`. For example,

```
JLabel message = new JLabel("Hello World!", JLabel.CENTER);
```

creates a label whose text is centered in the available space. You can change the text displayed in a label by calling the label's `setText()` method:

```
message.setText("Goodby World!");
```

Since the `JLabel` class is a subclass of `Component`, you can use methods such as `setForeground()` with labels. If you want the background color to have any effect, you should call `setOpaque(true)` on the label, since otherwise the `JLabel` might not fill in its background (depending on the look-and-feel). For example:

```
JLabel message = new JLabel("Hello World!");
message.setForeground(Color.red);    // Display red text...
message.setBackground(Color.black); // on a black background...
message.setFont(new Font("Serif",Font.BOLD,18)); // in a bold font.
message.setOpaque(true); // Make sure background is filled in.
```

The JCheckBox Class

A `JCheckBox` is a component that has two states: selected or unselected. The user can change the state of a check box by clicking on it. The state of a checkbox is represented by a boolean value that is `true` if the box is selected and `false` if the box is unselected. A checkbox has a label, which is specified when the

box is constructed:

```
JCheckBox showTime = new JCheckBox("Show Current Time");
```

Usually, it's the user who sets the state of a `JCheckBox`, but you can also set the state in your program. The current state of a checkbox is set using its `setSelected(boolean)` method. For example, if you want the checkbox `showTime` to be checked, you would say `showTime.setSelected(true);`. To uncheck the box, say `showTime.setSelected(false);`. You can determine the current state of a checkbox by calling its `isSelected()` method, which returns a boolean value.

In many cases, you don't need to worry about events from checkboxes. Your program can just check the state whenever it needs to know it by calling the `isSelected()` method. However, a checkbox does generate an event when its state changes, 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 of a checkbox, you must register an `ActionListener` with the checkbox. (Note that if you change the state by calling the `setSelected()` method, no `ActionEvent` is generated. However, there is another method in the `JCheckBox` class, `doClick()`, which simulates a user click on the checkbox and does generate an `ActionEvent`.)

When handling an `ActionEvent`, you can call `evt.getSource()` in the `actionPerformed()` method to find out which object generated the event. (Of course, if you are only listening for events from one component, you don't even have to do this.) The returned value is of type `Object`, but you can type-cast it to another type if you want. Once you know the object that generated the event, you can ask the object to tell you its current state. For example, if you know that the event had to come from one of two checkboxes, `cb1` or `cb2`, then your `actionPerformed()` method might look like this:

```
public void actionPerformed(ActionEvent evt) {
    Object source = evt.getSource();
    if (source == cb1) {
        boolean newState = ((JCheckBox)cb1).getSelected();
        ... // respond to the change of state
    }
    else if (source == cb2) {
        boolean newState = ((JCheckBox)cb2).getSelected();
        ... // respond to the change of state
    }
}
```

Alternatively, you can use `evt.getActionCommand()` to retrieve the action command associated with the source. For a `JCheckBox`, the action command is, by default, the label of the checkbox.

The `JRadioButton` and `ButtonGroup` Classes

Closely related to checkboxes are **radio buttons**. Radio buttons occur in groups. At most one radio button in a group can be selected at any given time. In Java, a radio button is represented by an object of type `JRadioButton`. When used in isolation, a `JRadioButton` acts just like a `JCheckBox`, and it has the same methods and events. Ordinarily, however, a `JRadioButton` is used in a group. A group of radio buttons is represented by an object belonging to the class `ButtonGroup`. A `ButtonGroup` is *not* a component and does not itself have a visible representation on the screen. A `ButtonGroup` works behind the scenes to organize a group of radio buttons, so that at most one button in the group can be selected at any given time.

To use a group of radio buttons, you must create a `JRadioButton` object for each button in the group, and you must create one object of type `ButtonGroup` to organize the individual buttons into a group.

Each `JRadioButton` must be added individually to some container, so that it will appear on the screen. (A `ButtonGroup` plays no role in the placement of the buttons on the screen.) Each `JRadioButton` must also be added to the `ButtonGroup`, which has an `add()` method for this purpose. If you want one of the buttons to be selected at start-up, you can call `setSelected(true)` for that button. If you don't do this, then none of the buttons will be selected until the user clicks on one of them.

As an example, here is how you could set up a set of radio buttons that can be used to select a color:

```
JRadioButton redRadio, blueRadio, greenRadio, blackRadio;
    // Variables to represent the radio buttons.
    // These should probably be instance variables, so
    // that they can be used throughout the program.

ButtonGroup colorGroup = new ButtonGroup();

redRadio = new JRadioButton("Red"); // Create a button.
colorGroup.add(redRadio);           // Add it to the group.

blueRadio = new JRadioButton("Blue");
colorGroup.add(blueRadio);

greenRadio = new JRadioButton("Green");
colorGroup.add(greenRadio);

blackRadio = new JRadioButton("Black");
colorGroup.add(blackRadio);

redRadio.setSelected(true); // Make an initial selection.
```

The individual buttons must still be added to a container if they are to appear on the screen. If you want to respond immediately when the user clicks on one of the radio buttons, you should register an `ActionListener` for each button. Here is an applet that demonstrates this. When you click one of the radio buttons, the background color of the label is changed:

(Applet "RadioButtonDemo" would be displayed here
if Java were available.)

The source code for the applet is in the file [RadioButtonDemo.java](#). Just as for checkboxes, it is not always necessary to register listeners for radio buttons. In many cases, you can simply check the state of each button when you need to know it, using the `isSelected()` method.

The JComboBox Class

The `JComboBox` class represents another way of letting the user select one option from a list of options. But in this case, the options are presented as a kind of pop-up menu, and only the currently selected option is visible on the screen. The painting applet at the end of [Section 6.6](#) used a `JComboBox` for selecting a color.

When a `JComboBox` object is first constructed, it initially contains no items. An item is added to the bottom of the menu by calling its instance method, `addItem(str)`, where `str` is a string that will be displayed. (In fact, you can add any type of object to a `JComboBox`. The `toString()` method of the object is called to determine what string to display.)

For example, the following code will create an object of type `JComboBox` that contains the options Red, Blue, Green, and Black:

```
JComboBox colorChoice = new JComboBox();
colorChoice.addItem("Red");
colorChoice.addItem("Blue");
colorChoice.addItem("Green");
colorChoice.addItem("Black");
```

You can call the `getSelectedIndex()` method of a `JComboBox` to find out which item is currently selected. This method returns an integer that gives the position of the selected item in the list, where the items are numbered starting from zero. Alternatively, you can call `getSelectedItem()` to get the selected item itself. (This method returns a value of type `Object`.) You can change the selection by calling the method `setSelectedIndex(n)`, where `n` is an integer giving the position of the item that you want to select.

The most common way to use a `JComboBox` menu is to call its `getSelectedIndex()` method when you have a need to know which item is currently selected. However, like other components that we have seen, `JComboBox` components generate `ActionEvents`. You can register an `ActionListener` with the `JComboBox` if you want to respond to such events as they occur.

`JComboBoxes` have a nifty feature, which is probably not all that useful in practice. You can make a `JComboBox` "editable" by calling its method `setEditable(true)`. If you do this, the user can edit the selection by clicking on the `JComboBox` and typing. This allows the user to make a selection that is not in the pre-configured list that you provide. (The "Combo" in the name "JComboBox" refers to the fact that it's a kind of combination of menu and text-input box.) If the user has edited the selection in this way, then the `getSelectedIndex()` method will return the value `-1`, and `getSelectedItem()` will return the string that the user typed. An `ActionEvent` is triggered if the user presses return in the `JComboBox`.

The JSlider Class

A `JSlider` provides a way for the user to select an integer value from a range of possible values. The user does this by dragging a "knob" along a bar. A slider can, optionally, be decorated with tick marks and with labels. This demonstration applet shows three sliders with different decorations and with different ranges of values:

(Applet "SliderDemo" would be displayed here
if Java were available.)

In this applet, the second slider is decorated with ticks, and the third one is decorated with labels. It's possible for a single slider to have both types of decorations.

The most commonly used constructor for `JSliders` specifies the start and end of the range of values for the slider and its initial value when it first appears on the screen:

```
JSlider(int minimum, int maximum, int value)
```

If the parameters are omitted, the values 0, 100, and 50 are used. By default, a slider is horizontal, but you can make it vertical by calling its method `setOrientation(JSlider.VERTICAL)`. The current value of a `JSlider` can be read at any time with its `getValue()` method. This method returns a value of type `int`. If you want to change the value, you can do so with the method `setValue(n)`, which takes a parameter of type `int`.

If you want to respond immediately when the user changes the value of a slider, you can register a listener with the slider. `JSliders`, unlike other components we have seen, do not generate `ActionEvents`. Instead, they generate events of type `ChangeEvent`. `ChangeEvent` and related classes are defined in the package `javax.swing.event` rather than `java.awt.event`, so if you want to use `ChangeEvents`, you should import `javax.swing.event.*` at the beginning of your program.

You must also define some object to implement the `ChangeListener` interface, and you must register the change listener with the slider by calling its `addChangeListener()` method. A `ChangeListener` must provide a definition for the method:

```
void stateChanged(ChangeEvent evt)
```

This method will be called whenever the value of the slider changes. (Note that it will be called when you change the value with the `setValue()` method, as well as when the user changes the value.) In the `stateChanged()` method, you can call `evt.getSource()` to find out which object generated the event.

Using tick marks on a slider is a two-step process: Specify the interval between the tick marks, and tell the slider that the tick marks should be displayed. There are actually two types of tick marks, "major" tick marks and "minor" tick marks. You can have one or the other or both. Major tick marks are a bit longer than minor tick marks. The method `setMinorTickSpacing(i)` indicates that there should be a minor tick mark every `i` units along the slider. The parameter is an integer. (The spacing is in terms of values on the slider, not pixels.) For the major tick marks, there is a similar command, `setMajorTickSpacing(i)`. Calling these methods is not enough to make the tick marks appear. You also have to call `setPaintTicks(true)`. For example, the second slider in the above applet was created and configured using the commands:

```
slider2 = new JSlider();
slider2.addChangeListener(this);
slider2.setMajorTickSpacing(25);
slider2.setMinorTickSpacing(5);
slider2.setPaintTicks(true);
getContentPane().add(slider2);
```

Labels on a slider are handled similarly. You have to specify the labels and tell the slider to paint them. Specifying labels is a tricky business, but the `JSlider` class has a method to simplify it. Create a set of labels and add them to a slider named `sldr` with the command:

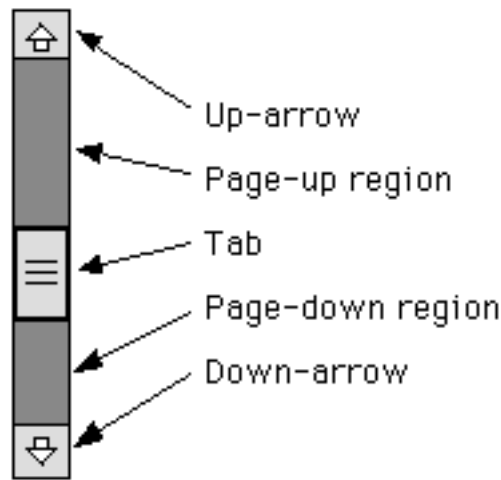
```
sldr.setLabelTable( sldr.createStandardLabels(i) );
```

where `i` is an integer giving the spacing between the labels. To arrange for the labels to be displayed, call `setPaintLabels(true)`. For example, the third slider in the above applet was created and configured with the commands:

```
slider3 = new JSlider(2000,2100,2002);
slider3.addChangeListener(this);
slider3.setLabelTable(slider3.createStandardLabels(50));
slider3.setPaintLabels(true);
getContentPane().add(slider3);
```

JScrollbar and JScrollPane

A `JScrollbar`, like a `JSlider`, allows the user to select an integer value from a range of values. A scroll bar, however, is generally used to control the scrolling of another component such as the text in a text editor. A scroll bar can be either horizontal or vertical. It has five parts:



The position of the tab specifies the currently selected value. The user can move the tab by dragging it or by clicking on any of the other parts of the scrollbar. The size of the tab tells what portion of a scrolling region is currently visible. It is actually the position of the bottom or left edge of the tab that represents the currently selected value.

A scrollbar has four associated integer values:

- **min**, which specifies the starting point of the range of values represented by the scrollbar, corresponding to the left or bottom edge of the bar
- **max**, which specifies the end point of the range of values, corresponding to the right or top edge of the bar
- **visible**, which specifies the size of the tab
- **value**, which gives the currently selected value, somewhere in the range between **min** and **(max - visible)**.

Note that the largest possible value is **(max - visible)**, not **max**, since the **value** represents the position of the left or bottom edge of the tab. The largest possible value allows space for the tab, whose size is given by **visible**.

The four values can be specified when the scrollbar is created. The constructor takes the form

```
JScrollbar(int orientation, int value, int visible, int min, int max);
```

The **orientation**, which specifies whether the scrollbar is horizontal or vertical, must be one of the constants **JScrollbar.HORIZONTAL** or **JScrollbar.VERTICAL**. The **value** must be between **min** and **(max - visible)**. You can leave out all the **int** parameters to get a scrollbar with default values. You can set the **value** of the scrollbar at any time with the method **setValue(int)**. Similarly, the other values can be set with **setMinimum(int)**, **setMaximum(int)**, and **setVisibleAmount(int)**. You can also set all four values at once by calling:

```
void setValues(int value, int visible, int min, int max);
```

Methods **getValue()**, **getVisibleAmount()**, **getMinimum()** and **getMaximum()** are provided for reading the current values of each of these parameters.

The user can drag the tab or click elsewhere on the scrollbar. How far does the tab move when the user clicks on the up-arrow or down-arrow or in the page-up or page-down region of a scrollbar? The amount by which the **value** changes when the user clicks on the up-arrow or down-arrow is called the **unit increment**. The amount by which it changes when the user clicks in the page-up or page-down region is called the **block increment**. By default, both of these values are 1. They can be set using the methods:

```
void setUnitIncrement(int unitIncrement);
```

```
void setBlockIncrement(int blockIncrement);
```

Let's look at an example. Suppose that you want to use a very large drawing area, which is too large to fit on the screen. You might decide to display only part of the `JPanel` and to provide scroll bars to allow the user to scroll through the entire panel. Let's say that the actual panel is 1000 by 1000 pixels, and that you will show a 200-by-200 region of the panel at any one time. Let's look at how you would set up the vertical scroll bar. The horizontal bar would be essentially the same.

The `visible` of the scroll bar would be 200, since that is how many pixels would actually be displayed. The value of the scroll bar would represent the vertical coordinate of the pixel that is at the top of the display. (Whenever the `value` changes, you have to redraw the display.) The `min` would be 0, and the `max` would be 1000. The range of values that can be set on the scroll bar is from 0 to 800. (Remember that the largest possible value is the maximum minus the visible amount.)

The page increment for the scroll bar could be set to some value a little less than 200, say 190 or 175. Then, when the user clicks in the page-up or page-down region, the display will scroll by an amount almost equal to its size. The line increment could be left at 1, but it is likely that this would be too small since it represents a scrolling increment of just one pixel. A line increment of 15 would be better, since then the display would scroll by a more reasonable 15 pixels when the user clicks the up-arrow or down-arrow. (Of course, all these values would have to be reset if the display area is resized.)

A scroll bar generates an event of type `AdjustmentEvent` whenever the user changes the value of the scroll bar. The associated `AdjustmentListener` interface defines one method, `adjustmentValueChanged(AdjustmentEvent evt)`, which is called by the scroll bar to notify the listener that the value on the scroll bar has been changed. This method should repaint the display or make whatever other change is appropriate for the new value. The method `evt.getValue()` returns the current value on the scroll bar. If you are using more than one scroll bar and need to determine which scroll bar generated the event, use `evt.getSource()` to determine the source of the event.

Scrolling is complicated. Fortunately, Swing provides a class that can take care of many of the details. A `JScrollPane` is a component that provides scrolling for another component. That component is specified as a parameter to the constructor:

```
JScrollPane(Component content)
```

The content component appears in the center of the scroll pane. If it is too large to be displayed entirely, then horizontal and/or vertical scroll bars will appear that can be used for scrolling the content. You have to add the scroll pane to a container to make both the scroll pane and its content appear on the screen. This makes scrolling *very* easy, and makes it unusual to work with scroll bars directly.

A `JScrollPane` can use any component as content, but several Swing components, including the `JTextArea` that will be discussed below, are designed specifically to work with `JScrollPane`.

The JTextField and JTextArea Classes

`JTextFields` and `JTextAreas` are boxes where the user can type in and edit text. The difference between them is that a `JTextField` contains a single line of editable text, while a `JTextArea` can display multiple lines. It is also possible to set a `JTextField` or `JTextArea` to be read-only so that the user can read the text that it contains but cannot edit the text.

Both `JTextField` and `JTextArea` are subclasses of `javax.swing.text.JTextComponent`, which defines their common behavior. The `JTextComponent` class supports the idea of a **selection**. A selection is a subset of the characters in the `JTextComponent`, including all the characters from some starting position to some ending position. The selection is hilited on the screen. The user selects text by dragging the mouse over it. Some useful methods in class `JTextComponent` include the following. They

can, of course, be used for both JTextFields and JTextAreas.

```
void setText(String newText); // substitute newText
                               // for current contents
String getText(); // return a copy of the current contents
String getSelectedText(); // return the selected text
void select(int start, int end); // change the selection;
    // characters in the range start <= pos < end are
    // selected; characters are numbered starting from zero
void selectAll(); // select the entire text
int getSelectionStart(); // get starting point of selection
int getSelectionEnd(); // get end point of selection
void setEditable(boolean canBeEdited);
    // specify whether or not the text in the component
    // can be edited by the user
```

The requestFocus() method, inherited from JComponent, is also useful for text components. The constructor for a JTextField takes the form

```
JTextField(int columns);
```

where columns specifies the number of characters that should be visible in the text field. This is used to determine the preferred width of the text field. (Because characters can be of different sizes, the number of characters visible in the text field might not be exactly equal to columns.) You don't have to specify the number of columns; for example, you might use the text field in a context where it will expand to the maximum size available. In that case, you can use the constructor JTextField(), with no parameters. You can also use the following constructors, which specify the initial contents of the text field:

```
JTextField(String contents);
JTextField(String contents, int columns);
```

JTextField has a subclass, JPasswordField, which is identical except that it does not reveal the text that it contains. The characters in a JPasswordField are all displayed as asterisks (or some other fixed character). A password field is, obviously, designed to let the user enter a password without showing that password on the screen.

The constructors for a JTextArea are

```
JTextArea();
JTextArea(int lines, int columns);
JTextArea(String contents);
JTextArea(String contents, int lines, int columns);
```

The parameter lines specifies how many lines of text should be visible in the text area. This determines the preferred height of the text area. (The text area can actually contain any number of lines; the text area can be scrolled to reveal lines that are not currently visible.) It is common to use a JTextArea as the Center component of a BorderLayout. In that case, it isn't useful to specify the number of lines and columns, since the TextArea will expand to fill all the space available in the center area of the container.

The JTextArea class adds a few useful procedures to those inherited from JTextComponent:

```
void append(String text);
    // add the specified text at the end of the current
    // contents; line breaks can be inserted by using the
    // special character \n
void insert(String text, int pos);
    // insert the text, starting at specified position
void replaceRange(String text, int start, int end);
    // delete the text from position start to position end
```



```
//      and then insert the specified text in its place
void setLineWrap(boolean wrap);
// If wrap is true, then a line that is too long to be
// displayed in the text area will be "wrapped" onto
// the next line.  The default value is false.
```

A `JTextField` generates an `ActionEvent` when the user presses return while typing in the `JTextField`. The `JTextField` class includes an `addActionListener()` method that can be used to register a listener with a `JTextField`. In the `actionPerformed()` method, the `evt.getActionCommand()` method will return a copy of the text from the `JTextField`. It is also common to use a `JTextField` by checking its contents, when needed, with the `getText()` method. `JTextAreas` do not generate action events.

A `JTextArea` does not have scroll bars, but scroll bars can be added easily by putting the text area in a scroll pane:

```
JTextArea inputArea = new JTextArea();
JScrollPane scroller = new JScrollPane( inputArea );
```

The scroll bars will appear only when needed. Remember to add the scroll pane, not the text area, to a container.

Other Components

This section has introduced many, but not all, Swing components. We will look at menus and menu bars in [Section 5](#). Some Swing components will not be covered at all. These include:

- `JList` -- displays a list of items to the user, and allows the user to select one or several items from the list.
- `JTable` -- displays a two-dimensional table of items, and possibly allows the user to edit them.
- `JTree` -- displays hierarchical data in a tree-like structure.
- `JToolBar` -- holds a row of tools, such as icons and buttons. The user can drag the tool bar away from the window that contains it, and it becomes a separate, floating tool window.
- `JSplitPane` -- a container that displays two components. The user can drag the dividing line between the components to adjust their relative sizes.
- `JTabbedPane` -- a container that displays one of a set of panels. The user selects which panel is visible by clicking on a "tab" at the top of the pane.

[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Section 7.4

Programming with Components

THE TWO PREVIOUS SECTIONS described some raw materials that are available in the form of layout managers and standard GUI components. This section presents some programming examples that make use of those raw materials.

An Example with Text Input Boxes

As a first example, let's look at a simple calculator applet. This example demonstrates typical uses of `JTextFields`, `JButtons`, and `JLabels`, and it uses several layout managers. In the applet, you can enter two real numbers in the text-input boxes and click on one of the buttons labeled "+", "-", "*", and "/". The corresponding operation is performed on the numbers, and the result is displayed in a `JLabel` at the bottom of the applet. If one of the input boxes contains an illegal entry -- a word instead of a number, for example -- an error message is displayed in the `JLabel`.

(Applet "SimpleCalculator" would be displayed here
if Java were available.)

When designing an applet such as this one, you should start by asking yourself questions like: How will the user interact with the applet? What components will I need in order to support that interaction? What events can be generated by user actions, and what will the applet do in response? What data will I have to keep in instance variables to keep track of the state of the applet? What information do I want to display to the user? Once you have answered these questions, you can decide how to lay out the components. You might want to draw the layout on paper. At that point, you are ready to begin writing the program.

In the simple calculator applet, the user types in two numbers and clicks a button. The computer responds by doing a computation with the user's numbers and displaying the result. The program uses two `JTextField` components to get the user's input. The `JTextFields` do a lot of work on their own. They respond to mouse, focus, and keyboard events. They show blinking cursors when they are active. They collect and display the characters that the user types. The program only has to do three things with each `JTextField`: Create it, add it to the applet, and get the text that the user has input by calling its `getText()` method. The first two things are done in the applet's `init()` method. The third -- getting the user's input from the input boxes -- is done in an `actionPerformed()` method, which responds when the user clicks on one of the buttons. When a component is created in one method and used in another, as the input boxes are in this case, we need an instance variable to refer to it. In this case, I use two instance variables, `xInput` and `yInput`, of type `JTextField` to refer to the input boxes. The `JLabel` that is used to display the result is treated similarly: A `JLabel` is created and added to the applet in the `init()` method. When an answer is computed in the `actionPerformed()` method, the `JLabel`'s `setText()` method is used to display the answer in the label. I use an instance variable named `answer`, of type `JLabel`, to refer to the label.

The applet also has four `JButtons` and two more `JLabels`. (The two extra labels display the strings "x =" and "y =".) I use local variables rather than instance variables for these components because I don't need to refer to them outside the `init()` method.

The applet as a whole uses a `GridLayout` with four rows and one column. The bottom row is occupied by the `JLabel`, `answer`. The other three rows each contain several components. Each of the first three rows is filled by a `JPanel`, which has its own layout manager and contains several components. The row that contains the four buttons is a `JPanel` which uses a `GridLayout` with one row and four columns. The `JPanels` that contain the input boxes use `BorderLayouts`. The input box occupies the `Center` position of the `BorderLayout`, with a `JLabel` on the `West`. (This example shows that

BorderLayouts are more versatile than it might appear at first.) All the work of setting up the applet is done in its `init()` method:

```
public void init() {

    /* Since I will be using the content pane several times,
       declare a variable to represent it. Note that the
       return type of getContentPane() is Container. */

    Container content = getContentPane();

    /* Assign a background color to the applet and its
       content panel. This color will show through between
       components and around the edges of the applet. */

    setBackground(Color.gray);
    content.setBackground(Color.gray);

    /* Create the input boxes, and make sure that their background
       color is white. (They are likely to be white by default.) */

    xInput = new JTextField("0");
    xInput.setBackground(Color.white);
    yInput = new JTextField("0");
    yInput.setBackground(Color.white);

    /* Create panels to hold the input boxes and labels "x =" and
       "y = ". By using a BorderLayout with the JTextField in the
       Center position, the JTextField will take up all the space
       left after the label is given its preferred size. */

    JPanel xPanel = new JPanel();
    xPanel.setLayout(new BorderLayout());
    xPanel.add( new Label(" x = "), BorderLayout.WEST );
    xPanel.add(xInput, BorderLayout.CENTER);

    JPanel yPanel = new JPanel();
    yPanel.setLayout(new BorderLayout());
    yPanel.add( new Label(" y = "), BorderLayout.WEST );
    yPanel.add(yInput, BorderLayout.CENTER);

    /* Create a panel to hold the four buttons for the four
       operations. A GridLayout is used so that the buttons
       will all have the same size and will fill the panel.
       The applet serves as ActionListener for the buttons. */

    JPanel buttonPanel = new JPanel();
    buttonPanel.setLayout(new GridLayout(1,4));

    JButton plus = new JButton("+");
    plus.addActionListener(this);
    buttonPanel.add(plus);

    JButton minus = new JButton("-");
    minus.addActionListener(this);
    buttonPanel.add(minus);
```

```

        JButton times = new JButton("*");
        times.addActionListener(this);
        buttonPanel.add(times);

        JButton divide = new JButton("/");
        divide.addActionListener(this);
        buttonPanel.add(divide);

        /* Create the label for displaying the answer in red
           on a white background. The label is set to be
           "opaque" to make sure that the white background
           is painted. */

        answer = new JLabel("x + y = 0", JLabel.CENTER);
        answer.setForeground(Color.red);
        answer.setBackground(Color.white);
        answer.setOpaque(true);

        /* Set up the layout for the applet, using a GridLayout,
           and add all the components that have been created. */

        content.setLayout(new GridLayout(4,1,2,2));
        content.add(xPanel);
        content.add(yPanel);
        content.add(buttonPanel);
        content.add(answer);

        /* Try to give the input focus to xInput, which is the natural
           place for the user to start. */

        xInput.requestFocus();

    } // end init()

```

The action of the applet takes place in the `actionPerformed()` method. The algorithm for this method is simple:

```

get the number from the input box xInput
get the number from the input box yInput
get the action command (the name of the button)
if the command is "+"
    add the numbers and display the result in the answer label
else if the command is "-"
    subtract the numbers and display the result in the label
else if the command is "*"
    multiply the numbers and display the result in the label
else if the command is "/"
    divide the numbers and display the result in the label

```

There is only one problem with this. When we call `xInput.getText()` and `yInput.getText()` to get the contents of the input boxes, the results are `Strings`, not numbers. We need a method to convert a string such as "42.17" into the number that it represents. The standard class `Double` contains a static method, `Double.parseDouble(String)` for doing just that. So we can get the first number entered by the user with the commands: f

```
String xStr = xInput.getText();
x = Double.parseDouble(xStr);
```

where `x` is a variable of type `double`. Similarly, if we wanted to get an integer value from the string, `xStr`, we could use a static method in the standard `Integer` class:

`x = Integer.parseInt(xStr)`. This makes it easy to get numerical values from a `JTextField`, but one problem remains: We can't be sure that the user has entered a string that represents a legal real number. We could ignore this problem and assume that a user who doesn't enter a valid input shouldn't expect to get an answer. However, a more friendly program would notice the error and display an error message to the user. This requires using a "try...catch" statement, which is not covered until [Chapter 9](#) of this book. My program does in fact use a try...catch statement to handle errors, so you can get a preview of how it works. Here is the `actionPerformed()` method that responds when the user clicks on one of the buttons in the applet:

```
public void actionPerformed(ActionEvent evt) {
    // When the user clicks a button, get the numbers
    // from the input boxes and perform the operation
    // indicated by the button. Put the result in
    // the answer label. If an error occurs, an
    // error message is put in the label.

    double x, y; // The numbers from the input boxes.

    /* Get a number from the xInput JTextField. Use
       xInput.getText() to get its contents as a String.
       Convert this String to a double. The try...catch
       statement will check for errors in the String. If
       the string is not a legal number, the error message
       "Illegal data for x." is put into the answer and
       the actionPerformed() method ends. */

    try {
        String xStr = xInput.getText();
        x = Double.parseDouble(xStr);
    }
    catch (NumberFormatException e) {
        // The string xStr is not a legal number.
        answer.setText("Illegal data for x.");
        return;
    }

    /* Get a number from yInput in the same way. */

    try {
        String yStr = yInput.getText();
        y = Double.parseDouble(yStr);
    }
    catch (NumberFormatException e) {
        answer.setText("Illegal data for y.");
        return;
    }

    /* Perform the operation based on the action command
       from the button. Note that division by zero produces
       an error message. */
```

```

        String op = evt.getActionCommand();
        if (op.equals("+"))
            answer.setText( "x + y = " + (x+y) );
        else if (op.equals("-"))
            answer.setText( "x - y = " + (x-y) );
        else if (op.equals("*"))
            answer.setText( "x * y = " + (x*y) );
        else if (op.equals("/")) {
            if (y == 0)
                answer.setText("Can't divide by zero!");
            else
                answer.setText( "x / y = " + (x/y) );
        }
    } // end actionPerformed()

```

The complete source code for the applet can be found in the file [SimpleCalculator.java](#). (It contains very little in addition to the two methods shown above.)

An Example with Sliders

As a second example, let's look more briefly at another applet. In this example, the user manipulates three `JSliders` to set the red, green, and blue levels of a color. The value of each color level is displayed in a `JLabel`, and the color itself is displayed in a large rectangle:

(Applet "RGBColorChooser" would be displayed here
if Java were available.)

The layout manager for the applet is a `GridLayout` with one row and three columns. The first column contains a `JPanel`, which in turn contains the `JSliders`. This panel uses another `GridLayout`, with three rows and one column. The second column, which contains the `JLabels`, is similar. The third column contains the colored rectangle. The component in this column is a `JPanel` which contains no components. The displayed color is the background color of the `JPanel`. When the user changes the color, the background color of the panel is changed and the panel is repainted to show the new color. This is one of the few cases where an object of type `JPanel` is used without either making a subclass or adding components to it.

When the user changes the value on a `JSlider`, an event of type `ChangeEvent` is generated. In order to respond to such events, the applet implements the `ChangeListener` interface, which specifies the method `"public void stateChanged(ChangeEvent evt)"`. The applet registers itself to listen for change events from each slider. The applet has instance variables to refer to the sliders, the labels, and the color patch. Note that since the `ChangeEvent` and `ChangeListener` classes are defined in the package `javax.swing.event`, the command `"import javax.swing.event.*;"` is added to the beginning of the program.

Let's look at the code from the `init()` method for setting up one of the `JSliders`, `redSlider`:

```

        redSlider = new JSlider(0, 255, 0);
        redSlider.addChangeListener(this);

```

The first line constructs a horizontal slider whose value can range from 0 to 255. These are the possible values of the red level in a color. The initial value of the slider, which is specified by the third parameter to the constructor, is 0. The second line registers the applet (`"this"`) to listen for change events from the slider. The other two sliders are initialized in a similar way.

In the `stateChanged()` method, the applet must respond to the fact that the user has changed the value

of one of the sliders. The response is to read the values of all the sliders, set the labels to display those values, and change the color displayed on the color patch. (This is slightly lazy programming, since only one of the labels actually needs to be changed. However, there is no rule against setting the text of a label to the same text that it is already displaying.)

```
public void stateChanged(ChangeEvent evt) {
    // This is called when the user has changed the value on
    // one of the sliders. All the sliders are checked,
    // the labels are set to display the correct values, and
    // the color patch is set to correspond to the new color.
    int r = redSlider.getValue();
    int g = greenSlider.getValue();
    int b = blueSlider.getValue();
    redLabel.setText(" R = " + r);
    greenLabel.setText(" G = " + g);
    blueLabel.setText(" B = " + b);
    colorPatch.setBackground(new Color(r,g,b));
} // end stateChanged()
```

The complete source code can be found in the file [RGBColorChooser.java](#).

Custom Component Examples

Java's standard component classes are often all you need to construct a user interface. Sometimes, however, you need a component that Java doesn't provide. In that case, you can write your own component class, building on one of the components that Java does provide. We've already done this, actually, every time we've written a subclass of the `JPanel` class to use as a drawing surface. A `JPanel` is a blank slate. By defining a subclass, you can make it show any picture you like, and you can program it to respond in any way to mouse and keyboard events. Sometimes, if you are lucky, you don't need such freedom, and you can build on one of Java's more sophisticated component classes.

For example, suppose I have a need for a "stopwatch" component. When the user clicks on the stopwatch, I want it to start timing. When the user clicks again, I want it to display the elapsed time since the first click. The textual display can be done with a `JLabel`, but we want a `JLabel` that can respond to mouse clicks. We can get this behavior by defining a `StopWatch` component as a subclass of the `JLabel` class. A `StopWatch` 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 applet, set the applet to listen for mouse events on the label, and let the applet 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 is reusable 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 `StopWatch` class is not very hard to write. I need an instance variable to record the time when the user started the stopwatch. Times in Java are measured in milliseconds and are stored in variables of type `long` (to allow for very large values). In the `mousePressed()` method, I need to know whether the timer is being started or stopped, so I need another instance variable to keep track of this aspect of the component's state. There is one more item of interest: How do I know what time the mouse was clicked? The method `System.currentTimeMillis()` returns the current time. But there can be some delay between the time the user clicks the mouse and the time when the `mousePressed()` routine is called. I don't want to know the current time. I want to know the exact time when the mouse was pressed. When I wrote the `StopWatch` class, this need sent me on a search in the Java documentation. I found that if `evt` is an object of type `MouseEvent()`, then the function `evt.getWhen()` returns the time when the event

occurred. I call this function in the `mousePressed()` routine.

The complete `StopWatch` class is rather short:

```
import java.awt.event.*;
import javax.swing.*;

public class StopWatch extends JLabel implements MouseListener {

    private long startTime;    // Start time of timer.
                                //    (Time is measured in milliseconds.)

    private boolean running;    // True when the timer is running.

    public StopWatch() {
        // Constructor.
        super(" Click to start timer. ", JLabel.CENTER);
        addMouseListener(this);
    }

    public void mousePressed(MouseEvent evt) {
        // React when user presses the mouse by
        // starting or stopping the timer.
        if (running == false) {
            // Record the time and start the timer.
            running = true;
            startTime = evt.getWhen(); // Time when mouse was clicked.
            setText("Timing....");
        }
        else {
            // Stop the timer. Compute the elapsed time since the
            // timer was started and display it.
            running = false;
            long endTime = evt.getWhen();
            double seconds = (endTime - startTime) / 1000.0;
            setText("Time: " + seconds + " sec.");
        }
    }

    public void mouseReleased(MouseEvent evt) { }
    public void mouseClicked(MouseEvent evt) { }
    public void mouseEntered(MouseEvent evt) { }
    public void mouseExited(MouseEvent evt) { }

} // end StopWatch
```

Don't forget that since `StopWatch` is a subclass of `JLabel`, you can do anything with a `StopWatch` that you can do with a `JLabel`. You can add it to a container. You can set its font, foreground color, and background color. You can set the text that it displays (although this would interfere with its stopwatch function). You can even add a `Border` if you want.

Let's look at one more example of defining a custom component. Suppose that -- for no good reason whatsoever -- I want a component that acts like a `JLabel` except that it displays its text in mirror-reversed form. Since no standard component does anything like this, the `MirrorLabel` class is defined as a subclass of `JPanel`. It has a constructor that specifies the text to be displayed and a `setText()` method

that changes the displayed text. The `paintComponent()` method draws the text mirror-reversed, in the center of the component. This uses techniques discussed in [Section 1](#). Information from a `FontMetrics` object is used to center the text in the component. The reversal is achieved by using an off-screen image. The text is drawn to the off-screen image, in the usual way. Then the image is copied to the screen with the following command, where `OSC` is the variable that refers to the off-screen image:

```
g.drawImage(OSC, widthOfOSC, 0, 0, heightOfOSC,
            0, 0, widthOfOSC, heightOfOSC, this);
```

This is the version of `drawImage()` that specifies corners of destination and source rectangles. The corner `(0,0)` in `OSC` is matched to the corner `(widthOfOSC,0)` on the screen, while `(widthOfOSC,heightOfOSC)` is matched to `(0,heightOfOSC)`. This reverses the image left-to-right. Here is the complete class:

```
import java.awt.*;
import javax.swing.*;

public class MirrorLabel extends JPanel {

    // Constructor and methods meant for use public use.

    public MirrorLabel(String text) {
        // Construct a MirrorLabel to display the specified text.
        this.text = text;
    }

    public void setText(String text) {
        // Change the displayed text. Call revalidate
        // so that the layout of its container can be
        // recomputed.
        this.text = text;
        revalidate(); // Tells container that size might have changed.
        repaint();
    }

    public String getText() {
        // Return the string that is displayed by this component.
        return text;
    }

    // Implementation. Not meant for public use.

    private String text; // The text displayed by this component.

    private Image OSC;
        // An off-screen image holding the non-reversed text.

    private int widthOfOSC, heightOfOSC;
        // Current size of the off-screen image, if one exists.

    public void paintComponent(Graphics g) {
        // The paint method makes a new OSC, if necessary. It writes
        // a non-reversed copy of the string to the the OSC, then
        // reverses the OSC as it copies it to the screen.
        // (Note: color or font might have changed since the
        // last time paintComponent() was called, so I can't just
```

```

        // reuse the old image in the OSC.)
        if (OSC == null || getSize().width != widthOfOSC
            || getSize().height != heightOfOSC) {
            OSC = createImage(getSize().width, getSize().height);
            widthOfOSC = getSize().width;
            heightOfOSC = getSize().height;
        }
        Graphics OSG = OSC.getGraphics();
        OSG.setColor(getBackground());
        OSG.fillRect(0, 0, widthOfOSC, heightOfOSC);
        OSG.setColor(getForeground());
        OSG.setFont(getFont());
        FontMetrics fm = OSG.getFontMetrics(getFont());
        int x = (widthOfOSC - fm.stringWidth(text)) / 2;
        int y = (heightOfOSC + fm.getAscent() - fm.getDescent()) / 2;
        OSG.drawString(text, x, y);
        OSG.dispose();
        g.drawImage(OSC, widthOfOSC, 0, 0, heightOfOSC,
                    0, 0, widthOfOSC, heightOfOSC, null);
    } // end paintComponent()

    public Dimension getPreferredSize() {
        // Compute a preferred size that will hold the string plus
        // a border of 5 pixels.
        FontMetrics fm = getFontMetrics(getFont());
        return new Dimension(fm.stringWidth(text) + 10,
                              fm.getAscent() + fm.getDescent() + 10);
    }

} // end class MirrorLabel

```

This class defines the method "public Dimension getPreferredSize()". This method is called by a layout manager when it wants to know how big the component would like to be. Standard components come with a way of computing a preferred size. For a custom component based on a JPanel, it's a good idea to provide a custom preferred size. As I mentioned in [Section 1](#), every component has a method `setPreferredSize()` that can be used to set the preferred size of the component. For our MirrorLabel component, however, the preferred size depends the font and the text of the component, and these can change from time to time. We need a way to compute a preferred size on demand, based on the current font and text. That's what we do by defining a `getPreferredSize()` method. The system calls this method when it wants to know the preferred size of the component. In response, we can compute the preferred size based on the current font and text.

The Stopwatch and MirrorLabel class define components. Components don't stand on their own. You have to add them to an applet or other container. Here is an applet that demonstrates a MirrorLabel and a Stopwatch component:

(Applet "ComponentTest" would be displayed here
if Java were available.)

The source code for this applet is in the file [ComponentTest.java](#). The applet uses a FlowLayout, so the components are not arranged very neatly. The applet also contains a button, which is there to illustrate another fine point of programming with components. If you click the button labeled "Change Text in this Applet", the text in all the components will be changed. You can also click on the "Timing..." label to start and stop the Stopwatch. When you do any of these things, you will notice that the components will be rearranged to take the new sizes into account. This is known as "validating" the container. This is done

automatically when a standard component changes in some way that requires a change in preferred size or location. This may or may not be the behavior that you want. (Validation doesn't always cause as much disruption as it does in this applet. For example, in a `GridLayout`, where all the components are displayed at the same size, it will have no effect at all. I've chosen a `FlowLayout` for this example to make the effect more obvious.) A custom component such as `MirrorLabel` can call the `revalidate()` method to indicate that the container that contains the component should be validated. In the `MirrorLabel` class, `revalidate()` is called in the `setText()` method.

A Null Layout Example

As a final example, we'll look at an applet that does not use a layout manager. If you set the layout manager of a container to be `null`, then you assume complete responsibility for positioning and sizing the components in that container. For an applet, you can remove the layout manager with the command:

```
getContentPane().setLayout(null);
```

If `comp` is any component, then the statement

```
comp.setBounds(x, y, width, height);
```

puts the top left corner of the component at the point (x, y) , measured in the coordinated system of the container that contains the component, and it sets the width and height of the component to the specified values. You should only set the bounds of a component if the container that contains it has a null layout manager. In a container that has a non-null layout manager, the layout manager is responsible for setting the bounds, and you should not interfere with its job.

Assuming that you have set the layout manager to `null`, you can call the `setBounds()` method any time you like. (You can even make a component that moves or changes size while the user is watching.) If you are writing an applet that has a known, fixed size, then you can set the bounds of each component in the applet's `init()` method. That's what done in the following applet, which contains four components: two buttons, a label, and a panel that displays a checkerboard pattern. This applet doesn't do anything useful. The buttons just change the text in the label.

(Applet "NullLayoutDemo" would be displayed here
if Java were available.)

In the `init()` method of this applet, the components are created and added to the applet. Then the `setBounds()` method of each component is called to set the size and position of the component:

```
public void init() {

    getContentPane().setLayout(null); // I will do the layout myself!

    getContentPane().setBackground(new Color(0,150,0));
                                   // Set a dark green background.

    /* Create the components and add them to the content pane.  If you
       don't add them to the a container, they won't appear, even if
       you set their bounds! */

    board = new Checkerboard();
           // (Checkerboard is defined later in this class.)
    getContentPane().add(board);

    newGameButton = new JButton("New Game");
    newGameButton.addActionListener(this);
```

```

        getContentPane().add(newGameButton);

        resignButton = new JButton("Resign");
        resignButton.addActionListener(this);
        getContentPane().add(resignButton);

        message = new JLabel("Click \"New Game\" to begin a game.",
                                JLabel.CENTER);
        message.setForeground( new Color(100,255,100) );
        message.setFont(new Font("Serif", Font.BOLD, 14));
        getContentPane().add(message);

        /* Set the position and size of each component by calling
           its setBounds() method. */

        board.setBounds(20,20,164,164);
        newGameButton.setBounds(210, 60, 120, 30);
        resignButton.setBounds(210, 120, 120, 30);
        message.setBounds(0, 200, 330, 30);

        /* Add a border to the content pane.  Since the return
           type of getContentPane() is Container, not JComponent,
           getContentPane() must be type-cast to a JComponent
           in order to call the setBorder() method.  Although I
           think the content pane is always, in fact, a JPanel,
           to be safe I test that the return value really is
           a JComponent. */

        if (getContentPane() instanceof JComponent) {
            ((JComponent)getContentPane()).setBorder(
                BorderFactory.createEtchedBorder());
        }
    } // end init();

```

It's reasonably easy, in this case, to get an attractive layout. It's much more difficult to do your own layout if you want to allow for changes of size. In that case, you have to respond to changes in the container's size by recomputing the sizes and positions of all the components that it contains. If you want to respond to changes in a container's size, you can register an appropriate listener with the container. Any component generates an event of type `ComponentEvent` when its size changes (and also when it is moved, hidden, or shown). You can register a `ComponentListener` with the container and respond to size change events by recomputing the sizes and positions of all the components in the container. Consult a Java reference for more information about `ComponentEvents`. However, my real advice is that if you want to allow for changes in the container's size, try to find a layout manager to do the work for you.

[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Section 7.5

Menus and Menubars

ANY USER OF A GRAPHICAL USER INTERFACE is accustomed to selecting commands from menus, which can be found in a menu bar at the top of a window (or sometimes at the top of a screen). In Java, menu bars, menus, and the items in menus are `JComponents`, just like all the other Swing components. Java makes it easy to add a menu bar to a `JApplet` or, as we will see in [Section 7](#), to a `JFrame`. Here is a sample applet that uses menus:

(Applet "ShapeDrawWithMenus" would be displayed here
if Java were available.)

This is a much improved version of the `ShapeDraw` applet from [Section 5.4](#). You can add shapes to the large white drawing area and drag them around. To add a shape, select one of the commands in the "Add" menu. The other menus allow you to control the properties of the shapes and set the background color of the drawing area.

This applet illustrates many ideas related to menus. There is a **menu bar**. A menu bar serves as a container for menus. In this case, there are three **menus** in the menu bar. The menus have titles: "Add", "Color", and "Options". When you click on one of these titles, the **menu items** in the menu appear. Each menu has an associated **mnemonic**, which is a character that is underlined in the name. Instead of clicking on the menu, you can select it by pressing the mnemonic key while holding down the ALT key. (This assumes that the applet has the keyboard focus.)

Once the menu has appeared, you can select an item in the menu by clicking on it, or by using the arrow keys to select the item and then pressing return. It is possible to assign mnemonics to individual items in a menu, but I haven't done that in this example. The commands in the "Add" menu and the "Clear" command do have **accelerators**. An accelerator is a key or combination of keys that can be pressed to invoke a menu item without ever opening the menu. The accelerator is shown in the menu, next to the name of the item. For example, the accelerator for the "Rectangle" command in the "Add" menu is "Ctrl-R". This means that you can invoke the command by holding down the Control key and pressing the R key. (Again, this assumes that the applet has the keyboard focus. The accelerators might not function at all in an applet. If so, you'll see how they work in [Section 7](#))

The commands in the "Color" menu act like a set of radio buttons. Only one item in the menu can be selected at a given time. The selected item in this menu determines the color of newly added shapes. Similarly, two of the commands in the "Options" menu act just like checkboxes. The first of these items determines whether newly added shapes will be large or small. The second determines whether newly added shapes will have a black border drawn around them.

The last item in the "Options" menu is actually another menu. This is called a **sub-menu**. When you select this item, the sub-menu will appear. Select an item from the sub-menu to set the background color of the drawing area.

This applet also demonstrates a **pop-up menu**. The pop-up menu appears when you click on one of the shapes in just the right way. The exact action you have to take depends on the look-and-feel and is called the **pop-up trigger**. The pop-up trigger is probably either clicking with the right mouse button, clicking with the middle mouse button, or clicking while holding down the Control key. The pop-up menu in this example contains commands for editing the shape on which you clicked.

In the rest of this section, we'll look at how all this can be programmed. If you would like to see the complete source code of the applet, you will find it in the file [ShapeDrawWithMenus.java](#). The source code is just over 600 lines long. The menus are created and configured in a very long `init()` method.

Menu Bars and Menus

A menu bar is just an object that belongs to the class `JMenuBar`. Since `JMenuBar` is a subclass of `JComponent`, a menu bar could actually be used anywhere in any container. In practice though, a `JMenuBar` is generally added to a top-level container such as a `JApplet`. This can be done in the `init()` method of a `JApplet` using commands of the form

```
JMenuBar menubar = new JMenuBar();
setMenuBar(menubar);
```

The applet's `setMenuBar()` method does *not* add the menu bar to the applet's content pane. The menu bar appears in a separate area, above the content pane.

A menu bar is a container for menus. The type of menu that can appear in a menu bar is an object belonging to the class `JMenu`. The constructor for a `JMenu` specifies a title for the menu, which appears in the menu bar. A menu is added to a menu bar using the menu bar's `add()` method. For example, the following commands will create a menu with title "Options" and add it to the `JMenuBar`, `menubar`:

```
JMenu optionsMenu = new JMenu("Options");
menubar.add(optionsMenu);
```

A mnemonic can be added to a `JMenu` using the menu's `addMnemonic()` method, which takes a parameter of type `char`. For example:

```
optionsMenu.setMnemonic('O');
```

A mnemonic provides a keyboard shortcut for the menu. If a mnemonic has been set for a menu, then the menu can be opened by pressing the specified character key while holding down the `ALT` key. If the mnemonic character appears in the title of the menu, it will be underlined. The mnemonic does not have to be the first character in the title. In fact, it doesn't have to appear in the title at all. Uppercase and lowercase letters are equivalent for mnemonics.

Note, by the way, that you can add a menu to a menu bar either before or after you have added menu items to the menu. You can add a menu to a menu bar even after it has appeared on the screen, but I found that I had to call `menubar.validate()` after adding the menu to get the menu to appear. You can remove a menu from a `menubar` by calling `menubar.remove(menu)`, but again, I found it necessary to call `menubar.validate()` after doing this if the `menubar` is already on the screen.

Menu Items, Sub-menus, and Separators

Each of the items in a menu is an object belonging to the class `JMenuItem`. A `JMenuItem` can be created with a constructor that specifies the text that appears in the menu, and it can be added to the menu using the menu's `add()` method. For example:

```
JMenuItem clear = new JMenuItem("Clear");
optionsMenu.add(clear); // where optionsMenu is of type JMenu
```

You can specify a mnemonic for the menu item as the second parameter to the constructor:

`new JMenuItem("Clear", 'C')`. The `JMenu` class also has an `add()` method that takes a `String` as parameter. This version of `add()` creates a new menu item with the given string as its text, and it adds that menu item to the menu. Furthermore, the menu item that was created is returned as the value of the method. This means that the two commands shown above can be abbreviated to the single command:

```
JMenuItem clear = optionsMenu.add("Clear");
```

A `JMenuItem` generates an `ActionEvent` when it is invoked by the user. If you want the menu item to

have some effect when it is invoked, you have to add an `ActionListener` to the menu item. For example, if `listener` is the object of type `ActionListener` that is to respond to the "Clear" command, you can say:

```
clear.addActionListener(listener);
```

Action events from `JMenuItems` can be processed in the same way as action events from `JButtons`: When the `actionPerformed()` method of the listener is called, the action command will be the text of the menu item, and the source of the event will be the menu item object itself.

In many cases, the only things you want to do with a menu item are add it to a menu and add an action listener to it. It's possible to do both of these with one command. For example:

```
optionsMenu.add("Clear").addActionListener(listener);
```

This funny looking line does the following: `optionsMenu.add("Clear")` creates creates a menu item and adds it to the menu, `optionsMenu`, and it returns the menu item as the value of the method call. Then the `addActionListener()` method is applied to the return value, that is, to the menu item that was just created.

The items in a menu are often separated into logical groups by horizontal lines drawn across the menu. The "Options" menu in the sample applet contains two such lines. You can add a separating line to the end of a `JMenu` by calling the menu's `addSeparator()` method. For example:

```
optionsMenu.addSeparator();
```

The `JMenu` class is actually defined as a subclass of `JMenuItem`, which means that you can add one menu to another. The menu that is added appears as a sub-menu in the menu to which it is added. The title of the sub-menu appears as an item in the main menu. When the user selects this item, the sub-menu appears. For example, in the applet at the top of this page, the "Background Color" sub-menu of the "Options" menu is created with the commands:

```
JMenu background = new JMenu("Background Color");
optionsMenu.add(background); // Add as sub-menu.
background.add("Red").addActionListener(canvas);
background.add("Green").addActionListener(canvas);
background.add("Blue").addActionListener(canvas);
background.add("Cyan").addActionListener(canvas);
background.add("Magenta").addActionListener(canvas);
background.add("Yellow").addActionListener(canvas);
background.add("Black").addActionListener(canvas);
background.add("Gray").addActionListener(canvas);
background.add("White").addActionListener(canvas);
```

Checkbox and Radio Button Menu Items

The `JMenuItem` class has two subclasses, `JCheckBoxMenuItem` and `JRadioButtonMenuItem`, that can be used to create menu items that serve as check boxes and radio buttons. Check boxes and radio buttons were covered in [Section 3](#) and just about everything that was said there applies here as well.

A `JCheckBoxMenuItem` can be in one of two states, either selected or unselected. The user changes the state by selecting the menu item. Just as with a `JCheckBox`, you can determine the state of a `JCheckBoxMenuItem` by calling its `isSelected()` method. You can set the state by calling the item's `setSelected(boolean)` method. You can register an `ActionListener` with a `JCheckBoxMenuItem`, if you want to respond immediately when the user changes the state. In many cases, however, you can just check the state at the point in your program where you need to know it. For example, one of the `JCheckBoxMenuItems` in the sample applet determines the size of the shapes that

are added to the drawing area. If the "Add Large Shapes" box is checked when a shape is added, then the shape will be large; if not, the shape will be small. There is no action listener in this case because nothing happens when the user selects the item (except that it changes state). When the user adds a new shape, the program calls `addLargeShapes.isSelected()` to determine which size to use. (`addLargeShapes` is the instance variable that refers to the `JCheckBoxMenuItem`.)

`JRadioButtonMenuItems` are almost always used in groups, where at most one of the radio buttons in the group can be selected at any given time. As with `JRadioButtons`, all the `JRadioButtonMenuItems` in a group are added to a `ButtonGroup`, which ensures that at most one of the items is selected. In the sample applet, for example, there are nine `JRadioButtonMenuItems` in the "Color" menu. These are represented by instance variables named `red`, `green`, `blue`, and so on. The code that creates the menu items and adds them both to the menu and to a button group look like this:

```
ButtonGroup colorGroup = new ButtonGroup();

red = new JRadioButtonMenuItem("Red");
shapeColorMenu.add(red);    // Add to menu.
colorGroup.add(red);        // Add to button group.

green = new JRadioButtonMenuItem("Green");
shapeColorMenu.add(green);
colorGroup.add(green);

blue = new JRadioButtonMenuItem("Blue");
shapeColorMenu.add(blue);
colorGroup.add(blue);
.
.
.
```

Initially, the "Red" item is selected. This is accomplished with the command `red.setSelected(true)`. There are no `ActionListeners` for the `JRadioButtonMenuItems`. When a new shape is added to the drawing area, the program checks the items in the "Color" menu to see what color is selected, and that color is used as the color of the new shape. This is done in an `addShape()` method using code that look like:

```
if (red.isSelected())
    shape.setColor(Color.red);
else if (green.isSelected())
    shape.setColor(Color.green);
else if (blue.isSelected())
    shape.setColor(Color.blue);
.
.
.
```

Note that `red`, `green`, and the other variables that represent the menu items in the "Color" menu must be defined as instance variables since they are initialized in the `init()` method of the applet and are also used in another method. The variable that represents the `ButtonGroup`, on the other hand, is just a local variable in the `init()` method, since it is not used in any other method.

Accelerators

A menu item in a `JMenu` can have an accelerator. The accelerator is a key, possibly with some modifiers such as `ALT` or `Control`, that the user can press to invoke the menu item without opening the menu. The menu item is processed in exactly the same way whether it is invoked with an accelerator, with a mnemonic, or with the mouse.

An accelerator can be described by a string that specifies the key to be pressed and any modifiers that must be held down while the key is pressed. Modifiers are specified by the words `shift`, `alt`, `ctrl`, and `meta`. These must be lower case. The key is specified by an upper case letter or by the name of certain special keys including: `HOME`, `END`, `DELETE`, `INSERT`, `LEFT`, `RIGHT`, `UP`, `DOWN`, `F1`, `F2`, The string that describes an accelerator consists of as many modifiers as you want, followed by any one key specification. You don't use the string directly to create an accelerator. The string is passed as a parameter to the static method `KeyStroke.getKeyStroke(String)`, which returns an object of type `KeyStroke`. The `KeyStroke` object can be used to add an accelerator to a `JMenuItem`. This is done by calling the `JMenuItem`'s `setAccelerator()` method, which requires a parameter of type `KeyStroke`. For example, the first menu item in the "Add" menu of the sample applet was created with the commands:

```
JMenuItem rect = new JMenuItem("Rectangle");
rect.setAccelerator( KeyStroke.getKeyStroke("ctrl R") );
```

The menu item will be invoked if the user holds down the `Control` key and presses the `R` key. Although it's unfortunate that you have to go through the `KeyStroke` class, it's really not all that complicated. Here are a few more examples of accelerators:

```
menuItem.setAccelerator( KeyStroke.getKeyStroke("shift ctrl S") );
// User must hold down both SHIFT and
// Control, while pressing the S key.

menuItem.setAccelerator( KeyStroke.getKeyStroke("HOME") );
// User can invoke the menu item just by pressing
// the HOME key.

menuItem.setAccelerator( KeyStroke.getKeyStroke("alt F4") );
// Pressing the F4 key while holding down the ALT key
// is equivalent to selecting the menu item. On a
// Macintosh, ALT refers to the Command key. Under
// Windows and Linux, this accelerator will probably
// be non-functional, since the operating system will
// intercept ALT-F4 and interpret it as a request to
// close the window.
```

Enabling and Disabling Menu Items

Since a `JMenuItem` is a `JComponent`, you can call `menuItem.setEnabled(false)` to disable a menu item and `menuItem.setEnabled(true)` to enable a menu item. A menu item that is disabled will appear "grayed out," and the user will not be able to select it, either with the mouse or with an accelerator. It's always a good idea to give the user visual feedback about the state of a program. Disabling a menu item when it doesn't make any sense to select it is one good way of doing this.

Pop-up Menus

A pop-up menu is an object belonging to the class `JPopupMenu`. It can be created with a constructor that has no parameters. Menu items, sub-menus, and separating lines can be added to a `JPopupMenu` in exactly the same way that they would be added to a `JMenu`. Menu items in a `JPopupMenu` generate `ActionEvents`, just as they would if they were in a `JMenu`. However, a `JPopupMenu` is not added to a menu bar. In fact, it is not added to any container at all. A `JPopupMenu` has a `show()` method that you can call to make it appear on the screen. Once it has appeared, the user can invoke an item in the menu in the usual way. When the user makes a selection, the menu disappears. The user can click outside the menu or hit the Escape key to dismiss the menu without making any selection from it.

The `show()` method takes three parameters. The first parameter is a `Component`. Since a `JPopupMenu` is not contained in any component, you have to tell it which component it will be associated with when it appears on the screen. The next two parameters of `show()` are integers that specify where the popup should appear on the screen. The integers give the coordinates of the point where the upper left corner of the popup menu will be located. The coordinates are specified in the coordinate system of the component that is provided as the first parameter to `show()`.

You can call `show()` any time you like. Usually, though, it's done in response to a mouse click. In that case, the first parameter to `show()` is generally the component on which the user clicked, and the next two parameters are the `x` and `y` coordinates where the mouse was clicked. (Actually, I usually use something like `x-10` and `y-2` so that the mouse position will be inside the popup menu, rather than exactly at the upper left corner. This tends to work better.)

Suppose, for example, that you want to show the menu when the user right-clicks on a component. You need to set up a mouse listener for the component and call the popup menu's `show()` method in the `mousePressed()` method of the listener. Let's say that `popup` is the variable of type `JPopupMenu` that refers to the popup menu and that `comp` is the variable of type `JComponent` that refers to the component. Then the `mousePressed()` method could be written as:

```
public void mousePressed(MouseEvent evt) {
    if (evt.isMetaDown()) { // This tests for a right-click
        int x = evt.getX(); // X-coord of mouse click
        int y = evt.getY(); // Y-coord of mouse click
        popup.show( comp, x-10, y-2 );
    }
}
```

If the component is also serving as the mouse listener, you could replace `popup.show(comp,x-10,y-2)` with `popup.show(this,x-10,y-2)`. Note that the only thing you do in response to the user's click is show the popup. The commands in the popup have to be handled elsewhere, such as in the `actionPerformed()` method of an `ActionListener` that has been registered to receive action events from the menu items in the popup menu.

This will work, but it's not the best style for handling popup menus. The problem is that different platforms have different standard techniques for calling up popup menus. Under Windows, the user expects a right-click to call up a menu. Under MacOS, the user would expect to hold down the Control key while clicking. If you want your program to work in a natural way on all platforms, you can call `evt.isPopupTrigger()` to determine whether a given mouse event is the proper "trigger" for calling up a popup menu in the current look-and-feel. Unfortunately, to do things right, you have to check for the popup trigger in both `mousePressed()` and in `mouseReleased()`, since a given look-and-feel might use either type of event as a trigger. So, the code for showing the popup becomes:

```
public void mousePressed(MouseEvent evt) {
    if (evt.isPopupTrigger()) {
        int x = evt.getX(); // X-coord of mouse click
```

```
        int y = evt.getY(); // Y-coord of mouse click
        popup.show( comp, x-10, y-2 );
    }

    public void mouseReleased(MouseEvent evt) {
        if (evt.isPopupTrigger()) {
            int x = evt.getX(); // X-coord of mouse click
            int y = evt.getY(); // Y-coord of mouse click
            popup.show( comp, x-10, y-2 );
        }
    }
}
```

In a program that uses dragging in addition to popup menus, the mouse event handling routines can become quite complicated. This is true for the sample applet at the top of this page. You can check the [source code](#) to see how it's done.

[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Section 7.6

Timers, Animation, and Threads

JAVA IS A MULTI-THREADED LANGUAGE, which means that several different things can be going on, in parallel. A **thread** is the basic unit of program execution. A thread executes a sequence of instructions, one after the other. When the system executes a stand-alone program, it creates a thread. (Threads are usually called **processes** in this context, but the differences are not important here.) The commands of the program are executed sequentially, from beginning to end, by this thread. The thread "dies" when the program ends. In a typical computer system, many threads can exist at the same time. At a given time, only one thread can actually be running, since the computer's Central Processing Unit (CPU) can only do one thing at a time. (An exception to this is a multi-processing computer, which has several CPUs. At a given time, every CPU can be executing a different thread.) However, the computer uses **time sharing** to give the illusion that several threads are being executed at the same time, "in parallel." Time sharing means that the CPU executes one thread for a while, then switches to another thread, then to another..., and then back to the first thread -- typically about 100 times per second. As far as users are concerned, the threads might as well be running at the same time.

To say that Java is a multi-threaded language means that a Java program can create one or more threads which will then run in parallel with the program. This is a fundamental, built-in part of the language, not an option or add-on like it is in some languages. Still, programming with threads can be tricky, and should be avoided unless it is really necessary. Ideally, even then, you can avoid using threads directly by using well-tested classes and libraries that someone has written for you.

Animation In Swing

One of the places where threads are used in GUI programming is to do animation. An animation is just a sequence of still images displayed on the screen one after the other. If the images are displayed quickly enough and if the changes from one image to the next are small enough, then the viewer will perceive continuous motion. To program an animation, you need some way of displaying a sequence of images.

A GUI program already has at least one thread, an **event-handling thread**, which detects actions taken by the user and calls appropriate subroutines in the program to handle each event. An animation, however, is not driven by user actions. It is something that happens by itself, as an independent process. In Java, the most natural way to accomplish this is to create a separate thread to run the animation. Before the introduction of the Swing GUI, a Java programmer would have to deal with this thread directly. This made animation much more complicated than it should have been. The good news in Swing is that it is no longer necessary to program directly with threads in order to do simple animations. The neat idea in Swing is to integrate animation with event-handling, so that you can program animations using the same techniques that are used for the rest of the program.

In Swing, an animation can be programmed using an object belonging to the class `javax.swing.Timer`. A `Timer` object can generate a sequence of events on its own, without any action on the part of the user. To program an animation, all your program has to do is create a `Timer` and respond to each event from the timer by displaying another frame in the animation. Behind the scenes, the `Timer` runs a separate thread which is responsible for generating the events, but you never need to deal with the thread directly.

The events generated by a `Timer` are of type `ActionEvent`. The constructor for a `Timer` specifies two things: The amount of time between events and an `ActionListener` that will be notified of each event:

```
Timer(int delayTime, ActionListener listener)
```

The listener should be programmed to respond to events from the `Timer` in its `actionPerformed()` method. The delay time between events is specified in milliseconds (where one second equals 1000 milliseconds). The actual delay time between two events can be longer than the requested delay time, depending on how long it takes to process the events and how busy the computer is with other things. In a typical animation, somewhere between ten and thirty frames should be displayed every second. These rates correspond to delay times between 100 and 33.

A `Timer` does not start running automatically when it is created. To make it run, you must call its `start()` method. A timer also has a `stop()` method, which you can call to make it stop generating events. If you have stopped a timer and want to start it up again, you can call its `restart()` method. (The `start()` method should be called only once.) None of these methods have parameters.

Let's look at an example. In the following applet, you can start an animation running by clicking the "Start" button. When you do this, the text on the button changes to "Stop", and you can stop the animation by clicking the button again. This is yet another applet that says "Hello World." The animation simply cycles the color of the message through all possible hues:

(Applet "HelloWorldSpectrum" would be displayed here
if Java were available.)

Here is part of the source code for this applet, omitting the definition of the nested class that defines the drawing surface:

```
public class HelloWorldSpectrum extends JApplet {

    Display display; // A JPanel belonging to a nested "Display"
                    // class; used for displaying "Hello World."
                    // It defines a method "setColor(Color)" for
                    // setting the color of the displayed message.

    JButton startStopButton; // The button that will be used to
                             // start and stop the animation.

    Timer timer; // The timer that drives the animation. A timer
                // is started when the user starts the animation.
                // Each time an ActionEvent is received from the
                // timer, the color of the message will change.
                // The value of this variable is null when the
                // animation is not in progress.

    int colorIndex; // This is be a number between 0 and 100 that
                   // will be used to determine the color. It will
                   // increase by 1 each time a timer event is
                   // processed.

    public void init() {
        // This is called by the system to initialize the applet.
        // It adds a button to the "south" position in the applet's
        // content pane, and it adds a display panel to the "center"
        // position so that it will fill the rest of the content pane.

        display = new Display();
        // The component that displays "Hello World".
    }
}
```



```

        getContentPane().add(display, BorderLayout.CENTER);
        // Adds the display panel to the CENTER position of the
        // JApplet's content pane.

        JPanel buttonBar = new JPanel();
        // This panel will hold the button and appears
        // at the bottom of the applet.
        buttonBar.setBackground(Color.gray);
        getContentPane().add(buttonBar, BorderLayout.SOUTH);

        startStopButton = new JButton("Start");
        buttonBar.add(startStopButton);

        startStopButton.addActionListener( new ActionListener() {
            // The action listener that responds to the
            // button starts or stops the animation. It
            // checks the value of timer to find out which
            // to do. Timer is non-null when the animation
            // is running, so if timer is null, the
            // animation needs to be started.
            public void actionPerformed(ActionEvent evt) {
                if (timer == null)
                    startAnimation();
                else
                    stopAnimation();
            }
        });

    } // end init()

    ActionListener timerListener = new ActionListener() {
        // Define an action listener to respond to events
        // from the timer. When an event is received, the
        // color of the display is changed.
        public void actionPerformed(ActionEvent evt) {
            colorIndex++; // A number between 0 and 100.
            if (colorIndex > 100)
                colorIndex = 0;
            float hue = colorIndex / 100.0F; // Between 0.0F and 1.0F.
            display.setColor( Color.getHSBColor(hue,1,1) );
        }
    };

    void startAnimation() {
        // Start the animation, unless it is already running.
        // We can check if it is running since the value of
        // timer is non-null when the animation is running.
        // (It should be impossible for this to be called
        // when an animation is already running... but it
        // doesn't hurt to check!)
        if (timer == null) {
            // Start the animation by creating a Timer that
            // will fire an event every 50 milliseconds, and
            // will send those events to timerListener.
            timer = new Timer(50, timerListener);
        }
    }

```

```

        timer.start(); // Make the time start running.
        startStopButton.setText("Stop");
    }
}

void stopAnimation() {
    // Stop the animation by stopping the timer, unless the
    // animation is not running.
    if (timer != null) {
        timer.stop(); // Stop the timer.
        timer = null; // Set timer variable to null, so that we
                     // can tell that the animation isn't running.
        startStopButton.setText("Start");
    }
}

public void stop() {
    // The stop() method of an applet is called by the system
    // when the applet is about to be stopped, either temporarily
    // or permanently. We don't want a timer running while
    // the applet is stopped, so stop the animation. (It's
    // harmless to call stopAnimation() if the animation is not
    // running.)
    stopAnimation();
}

.
.
.

```

This applet responds to `ActionEvents` from two sources: the button and the timer that drives the animation. I decided to use a different `ActionListener` object for each source. Each listener object is defined by an anonymous nested class. (It would, of course, be possible to use a single object, such as the applet itself, as a listener, and to determine the source of an event by calling `evt.getSource()`. However, Java programmers tend to be fond of anonymous classes.)

The applet defines methods `startAnimation()` and `stopAnimation()`, which are called when the user clicks the button. Each time the user clicks "Start", a new timer is created and started:

```

        timer = new Timer(50, timerListener);
        timer.start();

```

Remember that without `timer.start()`, the timer won't do anything at all. The constructor specifies a delay time of 50 milliseconds, so there should be about 20 action events from the timer every second. The second parameter is an `ActionListener` object. The timer events will be processed by calling the `actionPerformed()` method of this object. The `stopAnimation()` method calls `timer.stop()`, which ends the flow of events from the timer. The `startAnimation()` and `stopAnimation()` methods also change the text on the button, so that it reads "Stop" when the animation is running and "Start" when it is not running.

The `actionPerformed()` method in `timerListener` responds to a timer event by setting the color of the display. The color is computed from a number, `colorIndex` that is changed for each frame. (The color is specified as an "HSB" color. See [Section 6.3](#) for information on the HSB color system.)

JApplet's start() and stop() Methods

There is one other method in the `HelloWorldSpectrum` class that needs some explanation: the `stop()` method. Every applet has several methods that are meant to be called by the system at various times in the applet's life cycle. We have already seen that `init()` is called when the applet is first created, before it appears on the screen. Another method, `destroy()` is called just before the applet is destroyed, to give it a chance to clean things up. Two other applet methods, `start()` and `stop()`, are called by the system between `init()` and `destroy()`. The `start()` method is always called by the system just after `init()`, and `stop()` is always called just before `destroy()`. However, `start()` and `stop()` can also be called at other times. The reason is that an applet is not necessarily active for the whole time that it exists.

Suppose that you are viewing a Web page that contains an applet, and suppose you follow a link to another page. The applet still exists. It will be there if you hit your browser's Back button to return to the page that contains the applet. However, the page containing the applet is not visible, so the user can't see the applet or interact with it. The applet will not receive any events from the user, and since it is not visible, it will not be asked to paint itself. The system calls the applet's `stop()` method when user leaves the page that contains the applet. The system will call the applet's `start()` method if the user returns to that page. This lets the applet keep track of when it is active and when it is inactive. It might want to do this so that it can avoid using system resources when it is inactive.

In particular, an applet should probably not leave a timer running when it is inactive. The `HelloWorldSpectrum` applet defines the `stop()` method to call `stopAnimation()`, which, in turn, will stop the timer if it is running. When the applet is about to become inactive, the system will call `stop()`, and the animation -- if it was running -- will be stopped. You can try it, if you are reading this page in a Web browser: Start the animation running in the above applet, go to a different page, and then come back to this page. You should see that the animation has been stopped.

The animation in the `HelloWorldSpectrum` applet is started and stopped under the control of the user. In many cases, we want an animation to run for the entire time that an applet is active. In that case, the animation can be started in the applet's `start()` method and stopped in the applet's `stop()` method. It would also be possible to start the animation in the `init()` method and stop it in the `destroy()` method, but that would leave the animation running, uselessly, while the applet is inactive. In the following example, a message is scrolled across the page. It uses a timer which churns out events for the whole time the applet is active:

(Applet "ScrollingHelloWorld" would be displayed here
if Java were available.)

You can find the source code in the file [ScrollingHelloWorld.java](http://math.hws.edu/javanotes/c7/s6.html), but the relevant part here is the `start()` and `stop()` methods:

```
public void start() {
    // Called when the applet is being started or restarted.
    // Create a new timer, or restart the existing timer.
    if (timer == null) {
        // This method is being called for the first time,
        // since the timer does not yet exist.
        timer = new Timer(300, this); // (Applet listens for events.)
        timer.start();
    }
    else {
        timer.restart();
    }
}
```

```

public void stop() {
    // Called when the applet is about to be stopped.
    // Stop the timer.
    timer.stop();
}

```

These methods can be called several times during the lifetime of an applet. The first time `start()` is called, it creates and starts a timer. If `start()` is called again, the timer already exists, and the existing timer is simply restarted.

Other Useful Timer Methods

Although timers are most often used to generate a sequence of events, a timer can also be configured to generate a single event, after a specified amount of time. In this case, the timer is being used as an alarm which will signal the program after a specified amount of time has passed. To use a `Timer` object in this way, call `setRepeats(false)` after constructing it. For example:

```

Timer alarm = new Timer(5000, listener);
alarm.setRepeats(false);
alarm.start();

```

The timer will send one action event to the listener five seconds (5000 milliseconds) after it is started. You can cancel the alarm before it goes off by calling its `stop()` method.

Here's one final way to control the timing of events: When you start a repeating timer, the time until the first event is the same as the time between events. Sometimes, it would be convenient to have a longer or shorter delay before the first event. If you start a timer in the `init()` method of an applet, for example, you might want to give the applet some time to appear on the screen before receiving any events from the timer. You can set a separate delay for the first event by calling `timer.setInitialDelay(delay)`, where `timer` is the `Timer` and `delay` is specified in milliseconds as usual.

Using Threads

Although `Timers` can replace threads in some cases, there are times when direct use of threads is necessary. When a `Timer` is used, the processing is done in an event handler. This means that the processing must be something that can be done quickly. An event handler should always finish its work quickly, so that the system can move on to handle the next event. If an event handler runs for a long time, it will block other events from being processed, and the program will become unresponsive. So, in a GUI application, any computation or process that will take a long time to complete should be run in a separate thread. Then the event-handling thread can continue to run at the same time, responding to user actions as they occur.

As a short and incomplete introduction to threads, we'll look at one example. The example requires some explanation, but the main point for our discussion of threads is that it is a realistic example of a long computation that requires a separate thread. The example is based on the Mandelbrot set, a mathematical curiosity that has become familiar because it can be used to produce a lot of pretty pictures. The Mandelbrot set has to do with the following simple algorithm:

```

Start with a point (x,y) in the plane, where x and y are real numbers.
Let zx = x, and let zy = y.
Repeat the following:
    Replace (zx,zy) with ( zx*zx - zy*zy + x, 2*zx*zy + y )

```

The question is, what will happen to the point (zx, zy) as the loop is repeated over and over? The answer depends on the initial point (x, y) . For some initial points (x, y) , the point (zx, zy) will, sooner or later, move arbitrarily far away from the origin, $(0, 0)$. For other starting points, the point (zx, zy) will stay close to $(0, 0)$ no matter how many times you repeat the loop. The Mandelbrot set consists of the (x, y) points for which (zx, zy) stays close to $(0, 0)$ forever. This would probably not be very interesting, except that the Mandelbrot set turns out to have an incredibly intricate and quite pretty structure.

To get a pretty picture from the Mandelbrot set, we change the question, just a bit. Given a starting point (x, y) , we ask, how many steps does it take, up to some specified maximum number, before the point (zx, zy) moves some set distance away from $(0, 0)$? We then assign the point a color, depending on the number of steps. If we do this for each (x, y) , we get a kind of picture of the set. For a point in the Mandelbrot set, the count always reaches the maximum (since for such points, (zx, zy) *never* moves far away from zero). For other points, in general, the closer the point is to the Mandelbrot set, the more steps it will take.

With all that said, here is an applet that computes a picture of the Mandelbrot set. It will begin its computation when you press the "Start" button. (Eventually, the color of every pixel in the applet will be computed, but the applet actually computes the colors progressively, filling the applet with smaller and smaller blocks of color until it gets down to the single pixel level.) The applet represents a region of the plane with $-1.25 \leq x \leq 1.0$ and $-1.25 \leq y \leq 1.25$. The Mandelbrot set is colored purple. Points outside the set have other colors. Try it:

(Applet "Mandelbrot" would be displayed here
if Java were available.)

The algorithm for computing the colors in this applet is:

```
For square sizes 64, 32, 16, 8, 4, 2, and 1:
  For each square in the applet:
    Let (a,b) be the pixel coords of the center of the square.
    Let (x,y) be the real numbers corresponding to (a,b).
    Let (zx,zy) = (x,y).
    Let count = 0.
    Repeat until count is 80 or (zx,zy) is "big":
      Let new_zx = zx*zx - zy*zy + x.
      Let zy = 2*zx*zy + y.
      Let zx = new_zx.
      Let count = count + 1.
    Let color = Color.getHSBColor( count/100.0F, 0.0F, 0.0F )
    Fill the square with that color.
```

The point is that this is a *long* computation. When you click the "Start" button of the applet, the applet creates a separate thread to do this computation.

In Java, a thread is an object belonging to the class `java.lang.Thread`. The purpose of a thread is to execute a single subroutine from beginning to end. In the Mandelbrot applet, that subroutine implements the above algorithm. The subroutine for a thread is usually an instance method

```
public void run()
```

that is defined in an object that implements the interface named `Runnable`. The `Runnable` interface defines `run()` as its only method. The `Runnable` object is provided as a parameter to the constructor of the thread object. (It is also possible to define a thread by declaring a subclass of class `Thread`, and defining a `run()` method in the subclass, but it is more common to use a `Runnable` object to provide the `run()` method for a thread.) If `runnableObject` is an object that implements the `Runnable` interface, then a thread can be constructed with the command:

```
Thread runner = new Thread(runnableObject);
```

The job of this thread is to execute the `run()` method in `runnableObject`. Just as with a `Timer`, it is not enough to construct a thread. You must also start it running by calling its `start` method:

`runner.start()`. When this method is called, the thread will begin executing the `run()` method in the `runnableObject`, and it will do this in parallel with the rest of the program. When the subroutine ends, the thread will die, and it cannot be restarted or reused. There is no `stop` method for stopping a thread. (Actually, there is one, but it is deprecated, meaning that you are not supposed to call it.) If you want to be able to stop the thread, you need to provide some way of telling the thread to stop itself. I often do this with an instance variable named `running` that is visible both in the `run()` method and elsewhere in the program. When the program wants the thread to stop, it just sets the value of `running` to `false`. In the `run()` method, the thread checks the value of `running` regularly. If it sees that the value of `running` has become `false`, then the `run()` method should end. Here, for example, are the methods that the Mandelbrot applet uses to start and to stop the thread:

```
void startRunning() {
    // A simple method that starts the computational thread,
    // unless it is already running. (This should be
    // impossible since this method is only called when
    // the user clicks the "Start" button, and that button
    // is disabled when the thread is running.)
    if (running)
        return;
    runner = new Thread(this);
    // Creates a thread that will execute the run()
    // method in this class, which implements Runnable.
    running = true;
    runner.start();
}

void stopRunning() {
    // A simple method that is called to stop the computational
    // thread. This is done by setting the value of the
    // variable, running. The thread checks this value
    // regularly and will terminate when running becomes false.
    running = false;
}
```

There are a few more details of threads that you need to understand before looking at the `run()` method from the applet. First, on some platforms, once a thread starts running it grabs control of the CPU, and no other thread can run until it **yields** control. In Java, a thread can yield control, and allow other threads to run, by calling the static method `Thread.yield()`. I do this regularly in the `run()` method of the applet. If I did not do this, then, on some platforms, the computational thread would block the rest of the program from running. Another way for a thread to yield control is to go to **sleep** for a specified period of time by calling `Thread.sleep(time)`, where `time` is the number of milliseconds for which the thread will be inactive. The thread in a `Timer`, for example, sleeps between events, since it has nothing else to do.

Another issue concerns the use of threads with Swing. There is a rule in Swing: *Don't touch any GUI components, or any data used by them, except in the event-handling thread, that is, in event-handling methods or in `paintComponent()`*. The reason for this is that Swing is not **thread-safe**. If more than one thread plays with Swing's data structures, then the data can be corrupted. (This is done for efficiency. Making Swing thread-safe would slow it down significantly.) To solve this problem, Swing has a way for another thread to get the event-handling thread to run a subroutine for it. The subroutine must be a `run()` method in a `Runnable` object. When a thread calls the static method

```
void SwingUtilities.invokeLaterAndWait(Runnable runnableObject)
```

the `run()` method of `RunnableObject` will be executed in the event-handling thread, where it can safely do anything it wants to Swing components and their data. The `invokeAndWait` method, as the name indicates, does not return until the `run()` method has been executed. (The `invokeAndWait` method can produce an exception, and so must be called inside a `try...catch` statement. I will cover `try...catch` statements in [Chapter 9](#). You can ignore it for now.)

The `run()` method for the thread in the Mandelbrot applet uses `SwingUtilities.invokeLater()` to color a square on the screen. Here, finally, is that `run()` method (which will still take some work to understand):

```
public void run() {
    // This is the run method that is executed by the
    // computational thread. It draws the Mandelbrot
    // set in a series of passes of increasing resolution.
    // In each pass, it fills the applet with squares
    // that are colored to represent the Mandelbrot set.
    // The size of the squares is cut in half on each pass.

    startButton.setEnabled(false); // Disable "Start" button
    stopButton.setEnabled(true);   // and enable "Stop" button
                                   // while thread is running.

    int width = getWidth(); // Current size of this canvas.
    int height = getHeight();

    OSI = createImage(getWidth(),getHeight());
    // Create the off-screen image where the picture will
    // be stored, and fill it with black to start.
    OSG = OSI.getGraphics();
    OSG.setColor(Color.black);
    OSG.fillRect(0,0,width,height);

    for (size = 64; size >= 1 && running; size = size/2) {
        // Outer for loop performs one pass, filling
        // the image with squares of the given size.
        // The size here is given in terms of pixels.
        // Note that all loops end immediately if running
        // becomes false.
        double dx,dy; // Size of square in real coordinates.
        dx = (xmax - xmin)/width * size;
        dy = (ymax - ymin)/height * size;
        double x = xmin + dx/2; // x-coord of center of square.
        for (i = size/2; i < width+size/2 && running; i += size) {
            // First nested for loop draws one column of squares.
            double y = ymax - dy/2; // y-coord of center of square
            for (j = size/2; j < height+size/2 && running; j += size) {
                // Innermost for loop draws one square, by
                // counting iterations to determine what
                // color it should be, and then invoking the
                // "painter" object to actually draw the square.
                colorIndex = countIterations(x,y);
                try {
                    SwingUtilities.invokeLater(painter);
                }
                catch (Exception e) {
```



```

        }
        y -= dy;
    }
    x += dx;
    Thread.yield(); // Give other threads a chance to run.
}

running = false; // The thread is about to end, either
                  // because the computation is finished
                  // or because running has been set to
                  // false elsewhere. In the former case,
                  // we have to set running = false here
                  // to indicate that the thread is no
                  // longer running.

startButton.setEnabled(true); // Reset states of buttons.
stopButton.setEnabled(false);

} // end run()

```

You can find the full source code in the file [Mandelbrot.java](#).

There is a lot more to be said about threads. I will not cover it all in this book, but I will return to the topic in [Section 10.5](#).

[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Section 7.7

Frames and Dialogs

APPLETS ARE A FINE IDEA. It's nice to be able to put a complete program in a rectangle on a Web page. But more serious, large-scale programs have to run in their own windows, independently of a Web browser. In Java's Swing GUI library, an independent window is represented by an object of type `JFrame`. A stand-alone GUI application can open one or more `JFrames` to provide the user interface. It is even possible for an applet to open a frame. The frame will be a separate window from the Web browser window in which the applet is running. Any frame created by an applet includes a warning message such as "Warning: Insecure Applet Window." The warning is there so that you can always recognize windows created by applets. This is just one of the security restrictions on applets, which, after all, are programs that can be downloaded automatically from Web sites that you happen to stumble across without knowing anything about them.

Here is an applet that contains just one small button. When you click this "Launch ShapeDraw" button, a `JFrame` will be opened:

(Applet "ShapeDrawLauncher" would be displayed here
if Java were available.)

The frame that is created when you click the button is almost identical to the `ShapeDrawWithMenus` applet from [Section 5](#), and it is used in the same way. However, you can change the size of the window, and you can make the window go away by clicking its close box.

The window in this example belongs to a class named `ShapeDrawFrame`, which is defined as a subclass of `JFrame`. The structure of a `JFrame` is almost identical to a `JApplet`, and the programming is almost the same. In fact, only a few changes had to be made to the applet class, `ShapeDrawWithMenus`, to convert it from a `JApplet` to a `JFrame`. First, a frame does not have an `init()` method, so the initialization for `ShapeDrawFrame` is done in a constructor instead of in `init()`. In fact, the only real change necessary to convert a typical applet class into a frame class is to convert the applet's `init()` method into a constructor, and to add a bit of frame-specific initialization to the constructor. Everything that you learned about creating and programming GUI components and adding them to a content pane applies to frames as well. Menus are also handled in exactly the same way. So, we really only need to look at the additional programming that is necessary for frames.

One significant difference is that the size and location of an applet are determined externally to the applet, by the HTML code for a Web page and by the browser that displays the page. The size of a frame, on the other hand, has to be set by the frame itself or by the program that creates the frame. Often, the size is set in the frame's constructor. (If you forget to do this, the frame will have size zero and all you will see on the screen is a tiny border.) There are two methods in the `JFrame` class that can be used to set the size of a frame:

```
void setSize(int width, int height);
and
void pack();
```

Use `setSize()` if you know exactly what size the frame should be. The `pack()` method is more interesting. It should only be called after all GUI components have been added to the frame. Calling `pack()` will make the frame just big enough to hold all the components. It will determine the size of the frame by checking the preferred size of each of the components that it contains. (As mentioned in [Section 3](#), standard GUI components come with a preferred size. When you create your own drawing surface or custom component, you can set its preferred size by calling its `setPreferredSize()` method or by defining a `getPreferredSize()` method to compute the size.)

You can also set the location of the frame. This can be done with by calling the `JFrame` method:

```
void setLocation(int x, int y);
```

The parameters, `x` and `y` give the position of the upper left corner of the frame on the screen. They are given in terms of pixel coordinates on the screen as a whole, where the upper left corner of the screen is (0,0).

Creating a frame object does not automatically make a window appear on the screen. Initially, the window is invisible. You must make it visible by calling its method

```
void show();
```

This can be called at the end of the frame's constructor, but it can also be reasonable to leave it out. In that case, the constructor will produce an invisible window, and the program that creates the frame is responsible for calling its `show()` method.

A frame has a **title**, a string that appears in the title bar at the top of the window. This title can be provided as an argument to the constructor or it can be set by calling the method:

```
void setTitle(String title);
```

(In the `ShapeDrawFrame` class, I set the title of the frame to be "Shape Draw". I do this by calling a constructor in the superclass with the command:

```
super("Shape Draw");
```

at the beginning of the `ShapeDrawFrame` constructor.)

Now you know how to get a frame onto the screen and how to give it a title. There is still the matter of getting rid of the frame. You can hide a frame by calling its `hide()` method. If you do this, you can make it visible again by calling `show()`. If you are completely finished with the window, you can call its `dispose()` method to close the window and free the system resources that it uses. After calling `dispose()`, you should not use the frame object again. You also have to be prepared for the fact that the user can click on the window's close box to indicate that it should be closed. By default, the system will hide the window when the user clicks its close box. However, you can tell the system to handle this event differently. Sometimes, it makes more sense to dispose of the window or even to call `System.exit()` and end the program entirely. It is even possible, as we will see below, to set up a listener to listen for the event, and to program any response you want. You can set the default response to a click in a frame's close box by calling:

```
void setDefaultCloseOperation(int operation);
```

where the parameter, `operation` is one of the constants

- `JFrame.HIDE_ON_CLOSE` -- just hide the window, so it can be opened again
- `JFrame.DISPOSE_ON_CLOSE` -- destroy the window, but don't end the program
- `JFrame.EXIT_ON_CLOSE` -- terminate the Java interpreter by calling `System.exit()`
- `JFrame.DO_NOTHING_ON_CLOSE` -- no automatic response; your program is responsible for closing the window.

If a frame is opened by a main program, and if the program has only one window, it might make sense to use `EXIT_ON_CLOSE`. However, note that this option is illegal for a frame that is created by an applet, since an applet is not allowed to shut down the Java interpreter.

In case you are curious, here are the lines that I added to the end of the `ShapeDrawFrame` constructor:

```
setDefaultCloseOperation(EXIT_ON_CLOSE);
setLocation(20, 50);
setSize(550, 420);
show();
```

I also added a `main()` routine to the class. This means that it is possible to run `ShapeDrawFrame` as a stand-alone application. In a typical command-line environment, you would do this with the command:

```
java ShapeDrawFrame
```

It has been a while since we looked at a stand-alone program with a `main()` routine, and we have never seen a stand-alone GUI program. It's easy for a stand-alone program to use a graphical user interface -- all it has to do is open a frame. Since a `ShapeDrawFrame` makes itself visible when it is created, it is only necessary to create the frame object with the command `"new ShapeDrawFrame();"`. The complete main routine in this case looks like:

```
public static void main(String[] args) {
    new ShapeDrawFrame();
}
```

It might look a bit unusual to call a constructor without assigning the result to a variable, but it is perfectly legal and in this case we have no need to store the value. The main routine ends as soon as the frame is created, but the frame continues to exist and the program will continue running. Since the default close operation for the frame has been set to `EXIT_ON_CLOSE`, the frame will close and the program will be terminated when the user clicks the close box of the window. It might seem a bit odd to have this `main()` routine in the same class that defines `ShapeDrawFrame`, and in fact it could just as easily be in a separate class. But there is no real need to create an extra class, as long as you understand what is going on. When you type `"java ShapeDrawFrame"` on the command line, the system looks for a main routine in the `ShapeDrawFrame` class and executes it. If the main routine happens to create an object belonging to the same class, it's not a problem. It's just a command to be executed.

The source code for the frame class is in the file [ShapeDrawFrame.java](#). The applet, shown above, which opens a frame of this type is in [ShapeDrawLauncher.java](#).

We'll look at a few more fine points of programming with frames by looking at another example. In this case, it's a frame version of the `HighLowGUI2` applet from [Section 1](#). Click on this button to open the frame:

(Applet "HighLowLauncher" would be displayed here
if Java were available.)

The frame in this example is defined in the file [HighLowFrame.java](#). In many ways, this example is similar to the previous one, but there are several differences. You can resize the frame in the first example by dragging its border or corner, but if you try to resize the "High Low" frame, you will find that it is impossible. A frame is resizable by default. You can make it non-resizable by calling `setResizable(false)`. I do this in the constructor of the `HighLowFrame` class. Another difference shows up if you click the frame's close box. This will not simply close the window. Instead a new window will open to ask you whether you really want to close the `HighLow` frame. The new window is an example of a "dialog box". You will learn about dialog boxes later in this section. To proceed, you have to click a button in the dialog box. If you click on "Yes", the `HighLow` frame will be closed; if not, the frame will remain open. (This would be more useful if, for example, you wanted to give the user a chance to save some unsaved work before closing the window.) To get this behavior, I had to turn off the system's default handling of the close box with the command:

```
setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
```

and I had to program my own response instead. I did this by registering a **window listener** for the frame. When certain operations are performed on a window, the window generates an event of type `WindowEvent`. You can program a response to such events in the usual way: by registering a listening object of type `WindowListener` with the frame. The `JFrame` class has an `addWindowListener()` method for this purpose. The `WindowListener` must define seven

event-handling methods, including the poorly named

```

        public void windowClosing(WindowEvent evt)
and
        public void windowClosed(WindowEvent evt)

```

The `windowClosing` event is generated when the user clicks the close box of the window. The `windowClosed` event is generated when the window is actually being disposed. The other five methods in the `WindowListener` interface are more rarely used. Fortunately, you don't have to worry about them if you use the `WindowAdapter` class. The `WindowAdapter` class implements the `WindowListener` interface, but defines all the `WindowListener` methods to be empty. You can define a `WindowListener` by creating a subclass of `WindowAdapter` and providing definitions for just those methods that you actually need. In the `HighLowFrame` class, I need a listener to respond to the `windowClosing` event that is generated when the user clicks the close box. The listener is created in the constructor using an anonymous subclass of `WindowAdapter` with a command of the form:

```

addWindowListener( new WindowAdapter() {
    // This window listener responds when the user
    // clicks the window's close box by giving the
    // user a chance to change his mind.
    public void windowClosing(WindowEvent evt) {
        .
        . // Show the dialog box, and respond to it.
        .
    }
});

```

Another window listener is used in the little applet that appears on this page as the "Launch HighLow" button, above. This applet is defined in the file [HighLowLauncher.java](#). Note that when you click on the button to open the frame, the name of the button changes to "Close HighLow". You can close the frame by clicking the button again, as well as by clicking the frame's close box. When the frame is closed, for either reason, the name of the button changes back to "Launch HighLow". The question is, how does the applet know to change the button's name, when the user closes the frame by clicking its close box? That doesn't seem to have anything to do with the applet. The trick is to use an event listener. When the applet creates the frame, it also creates a `WindowListener` and registers it with the frame. This `WindowListener` is programmed to respond to the `windowClosed` event by changing the name of the button. This is a nice example of the sort of communication between objects that can be done with events. You can check the source code to see exactly how it's done.

Images in Applications

In [Section 1](#), we saw how to load an image file into an applet. The `JApplet` class has a method, `getImage()`, that can be used for this purpose. The `JFrame` class does not provide this method, so some other technique is needed for using images in frames.

The standard class `java.awt.Toolkit` makes it possible to load an image into a stand-alone application. A `Toolkit` object has a `getImage()` method for reading an `Image` from an image file. To use this method, you must first obtain a `Toolkit`, and you have to do this by calling the static method `Toolkit.getDefaultToolkit()`. (Any running GUI program already has a toolkit, which is used to perform various platform-dependent functions. You don't need to construct a new toolkit. `Toolkit.getDefaultToolkit()` returns a reference to the toolkit that already exists.) Once you have a toolkit, you can use its `getImage()` method to create an `Image` object from a file. This `getImage` method takes one parameter, which specifies the name of the file. For example:

```
Toolkit toolkit = getDefaultToolkit();
```

```
Image cards = toolkit.getImage( "smallcards.gif" );
```

or, in one line,

```
Image cards = Toolkit.getDefaultToolkit().getImage( "smallcards.gif" );
```

Once the Image has been created, it can be used in the same way, whether it has been created by an applet or a standalone application.

Note that an applet's `getImage()` method is used to load an image from a Web server, while a `Toolkit`'s `getImage()` loads an image from the same computer on which the program is running. An applet cannot, in general, use a `Toolkit` to load an image, since, for security reasons, an applet is not usually allowed to read files from the computer on which it is running.

The `HighLowFrame` example uses an image file named "smallcards.gif" for the cards that it displays. I designed `HighLowFrame` with a `main()` routine so that it can be run as a stand-alone application. (When it is run as an application, the "smallcards.gif" file must be in the same directory with the class files.) But a `HighLowFrame` can also be opened by an applet, as is done in the example on this page. When it is run as an application, the image file must be loaded by a `Toolkit`. When it is opened by an applet, the image file must be loaded by the `getImage()` method of the applet. How can this be handled? I decided to do it by making the `Image` object a parameter to the `HighLowFrame` constructor. The `Image` must be loaded before the frame is constructed, so that it can be passed as a parameter to the constructor. The `main()` routine in `HighLowFrame` does this using a `Toolkit`:

```
Image cards = Toolkit.getDefaultToolkit().getImage( "smallcards.gif" );
HighLowFrame game = new HighLowFrame(cards);
```

The `HighLowLauncher` applet, on the other hand, loads the image with its own `getImage()` method:

```
Image cards = getImage(getCodeBase(), "smallcards.gif");
highLow = new HighLowFrame(cards);
```

Dialogs

In addition to `JFrame`, there is another type of window in Swing. A **dialog box** is a type of window that is generally used for short, single purpose interactions with the user. For example, a dialog box can be used to display a message to the user, to ask the user a question, or to let the user select a color. In Swing, a dialog box is represented by an object belonging to the class `JDialog`.

Like a frame, a dialog box is a separate window. Unlike a frame, however, a dialog box is not completely independent. Every dialog box is associated with a frame (or another dialog box), which is called its **parent**. The dialog box is dependent on its parent. For example, if the parent is closed, the dialog box will also be closed. It is possible to create a dialog box without specifying a parent, but in that case a default frame is used or an invisible frame is created to serve as the parent.

Dialog boxes can be either **modal** or **modeless**. When a modal dialog is created, its parent frame is blocked. That is, the user will not be able to interact with the parent until the dialog box is closed. Modeless dialog boxes do not block their parents in the same way, so they seem a lot more like independent windows. In practice, modal dialog boxes are easier to use and are much more common than modeless dialogs. All the examples we will look at are modal.

Aside from having a parent, a `JDialog` can be created and used in the same way as a `JFrame`. However, we will not be using `JDialog` directly. Swing has many convenient methods for creating many common types of dialog boxes. For example, the `JColorChooser` class has the static method:

```
Color JColorChooser.showDialog(JFrame parent, Color initialColor)
```


When you call this method, a dialog box appears that allows the user to select a color. The first parameter specifies the parent of the dialog, or it can be `null`. When the dialog first appears, `initialColor` is selected. The dialog has a sophisticated interface that allows the user to change the selection. When the user presses an "OK" button, the dialog box closes and the selected color is returned as the value of the method. The user can also click a "Cancel" button or close the dialog box in some other way; in that case, `null` is returned as the value of the method. By using this predefined color chooser dialog, you can write one line of code that will let the user select an arbitrary color.

The following applet demonstrates a `JColorChooser` dialog and three other, simpler standard dialog boxes. When you click one of the buttons, a dialog box appears. The label at the top of the applet gives you some feedback about what is happening:

(Applet "SimpleDialogDemo" would be displayed here
if Java were available.)

The three simple dialogs in this applet are created by static methods in the class `JOptionPane`. This class includes many methods for making dialog boxes, but they are all variations on the three basic types shown here: a "message" dialog, a "confirm" dialog, and an "input" dialog. (The variations allow you to provide a title for the dialog box, to specify the icon that appears in the dialog, and to add other components to the dialog box. I will only cover the most basic forms here.)

A message dialog simply displays a message string to the user. The user (hopefully) reads the message and dismisses the dialog by clicking the "OK" button. A message dialog can be shown by calling the method:

```
void JOptionPane.showMessageDialog(JFrame parent, String message)
```

The parent, as usual, can be `null`. The message can be more than one line long. Lines in the message should be separated by newline characters, `\n`. New lines will not be inserted automatically, even if the message is very long.

An input dialog displays a question or request and lets the user type in a string as a response. You can show an input dialog by calling:

```
String JOptionPane.showInputDialog(JFrame parent, String question)
```

Again, the parent can be `null`, and the question can include newline characters. The dialog box will contain an input box and an "OK" button and a "Cancel" button. If the user clicks "Cancel", or closes the dialog box in some other way, then the return value of the method is `null`. If the user clicks "OK", then the return value is the string that was entered by the user. Note that the return value can be an empty string (which is not the same as a `null` value), if the user clicks "OK" without typing anything in the input box. If you want to use an input dialog to get a numerical value from the user, you will have to convert the return value into a number. A technique for doing this can be found in the first example in [Section 4](#).

Finally, a confirm dialog presents a question and three response buttons: "Yes", "No", and "Cancel". A confirm dialog can be shown by calling:

```
int JOptionPane.showConfirmDialog(JFrame parent, String question)
```

The return value tells you the user's response. It is one of the following constants:

- `JOptionPane.YES_OPTION` -- the user clicked the "Yes" button
- `JOptionPane.NO_OPTION` -- the user clicked the "No" button
- `JOptionPane.CANCEL_OPTION` -- the user clicked the "Cancel" button
- `JOptionPane.CLOSE_OPTION` -- the user closed the dialog in some other way.

I use a confirm dialog in the `HighLowFrame` example, earlier on this page. When the user clicks the close box of a `HighLowFrame`, I display a confirm dialog to ask whether the user really wants to quit. The frame will only be closed if the return value is `JOptionPane.YES_OPTION`.

By the way, it is possible to omit the Cancel button from a confirm dialog by calling one of the other methods in the `JOptionPane` class. Just call:

```
JOptionPane.showConfirmDialog(  
    parent, question, title, JOptionPane.YES_NO_OPTION )
```

The final parameter is a constant which specifies that only a "Yes" button and a "No" button should be used. The third parameter is a string that will be displayed as the title of the dialog box window.

If you would like to see how dialogs are created and used in the sample applet, you can find the source code in the file [SimpleDialogDemo.java](#).

End of Chapter 7

[[Next Chapter](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Programming Exercises For Chapter 7

THIS PAGE CONTAINS programming exercises based on material from [Chapter 7](#) of this [on-line Java textbook](#). Each exercise has a link to a discussion of one possible solution of that exercise.

Exercise 7.1: [Exercise 5.2](#) involved a class, [StatCalc.java](#), that could compute some statistics of a set of numbers. Write an applet that uses the `StatCalc` class to compute and display statistics of numbers entered by the user. The applet will have an instance variable of type `StatCalc` that does the computations. The applet 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 pressing return. There should be a "Clear" button that clears out all the data. This means creating a new `StatCalc` object and resetting the displays on the labels. My applet also has an "Enter" button that does the same thing as pressing the return key in the `TextField`. (Recall that a `TextField` generates an `ActionEvent` when the user presses return, so your applet should register itself to listen for `ActionEvents` from the `TextField`.) Here is my solution to this problem:

[See the solution!](#)

Exercise 7.2: Write an applet with a `TextArea` where the user can enter some text. The applet should have a button. When the user clicks on the button, the applet should count the number of lines in the user's input, the number of words in the user's input, and the number of characters in the user's input. This information should be displayed on three labels in the applet. 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 `TextArea`. The number of characters is just the length of this `String`. Lines in the `String` are separated by the new line character, '\n', so the number of lines is just the number of new line characters in the `String`, plus one. Words are a little harder to count. [Exercise 3.4](#) has some advice about finding the words in a `String`. Essentially, you want to count the number of characters that are first characters in words. Don't forget to put your `TextArea` in a `JScrollPane`. Scrollbars should appear when the user types more text than will fit in the available area. Here is my applet:

[See the solution!](#)

Exercise 7.3: The [RGBColorChooser](#) applet lets the user set the red, green, and blue levels in a color by manipulating sliders. Something like this could make a useful custom component. Such a component could be included in a program to allow the user to specify a drawing color, for example. Rewrite the `RGBColorChooser` as a component. Make it a subclass of `JPanel` instead of `JApplet`. Instead of doing the initialization in an `init()` method, you'll have to do it in a constructor. The component should have a method, `getColor()`, that returns the color currently displayed on the component. It should also have a method, `setColor(Color c)`, to set the color to a specified value. Both these methods would be useful to a program that uses your component.

In order to write the `setColor(Color c)` method, you need to know that if `c` is a variable of type `Color`, then `c.getRed()` is a function that returns an integer in the range 0 to 255 that gives the red level of the color. Similarly, the functions `c.getGreen()` and `c.getBlue()` return the blue and green

components.

Test your component by using it in a simple applet that sets the component to a random color when the user clicks on a button, like this one:

[See the solution!](#)

Exercise 7.4: In the Blackjack game [BlackjackGUI.java](#) from [Exercise 6.8](#), 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. Make this change in the Blackjack program. This applet uses a nested class, `BlackjackCanvas`, to represent the board. You'll have to do most of your work in that class. In order to manipulate the buttons, you will have to use instance variables to refer to the buttons.

I strongly advise writing a subroutine that can be called whenever it is necessary to set the value of the `gameInProgress` variable. Then the subroutine can take responsibility for enabling and disabling the buttons. Recall that if `btn` is a variable of type `JButton`, then `btn.setEnabled(false)` disables the button and `btn.setEnabled(true)` enables the button.

[See the solution!](#) [A working applet can be found [here](#).]

Exercise 7.5: Building on your solution to the preceding exercise, make it possible for the user to place bets on the Blackjack game. When the applet starts, give the user \$100. Add a `JTextField` to the strip of controls along the bottom of the applet. The user can enter the bet in this `JTextField`. When the game begins, check the amount of the bet. You should do this when the game begins, not when it ends, because several errors can occur: The contents of the `JTextField` might not be a legal number. The bet that the user places might be more money than the user has, or it might be ≤ 0 . You should detect these errors and show an error message instead of starting the game. The user's bet should be an integral number of dollars. You can convert the user's input into an integer, and check for illegal, non-numeric input, with a `try...catch` statement of the form

```
try {
    betAmount = Integer.parseInt( betInput.getText() );
}
catch (NumberFormatException e) {
    . . . // The input is not a number.
        // Respond by showing an error message and
        // exiting from the doNewGame() method.
}
```

It would be a good idea to make the `JTextField` uneditable while the game is in progress. If `betInput` is the `JTextField`, you can make it editable and uneditable by the user with the commands `betAmount.setEditable(true)` and `betAmount.setEditable(false)`.

In the `paintComponent()` method, you should include commands to display the amount of money that the user has left.

There is one other thing to think about: The applet should not start a new game when it is first created. The user should have a chance to set a bet amount before the game starts. So, in the constructor for the canvas class, you should not call `doNewGame()`. You might want to display a message such as "Welcome to Blackjack" before the first game starts.

[See the solution!](#) [A working applet can be found here.]

Exercise 7.6: The `StopWatch` component from [Section 7.4](#) displays the text "Timing..." when the stop watch is running. It would be nice if it displayed the elapsed time since the stop watch was started. For that, you need to create a `Timer`. Add a `Timer` to the original source code, [StopWatch.java](#), to display the elapsed time in seconds. Create the timer in the `mousePressed()` routine when the stop watch is started. Stop the timer in the `mousePressed()` routine when the stop watch is stopped. The elapsed time won't be very accurate anyway, so just show the integral number of seconds. You only need to set the text a few times per second. For my `Timer` method, I use a delay of 100 milliseconds for the timer. Here is an applet that tests my solution to this exercise:

[See the solution!](#)

Exercise 7.7: The applet at the end of [Section 7.7](#) shows animations of moving symmetric patterns that look something like the image in a kaleidoscope. Symmetric patterns are pretty. Make the `SimplePaint3` applet do symmetric, kaleidoscopic patterns. As the user draws a figure, the applet should be able to draw reflected versions of that figure to make symmetric pictures.

The applet will have several options for the type of symmetry that is displayed. The user should be able to choose one of four options from a `JComboBox` menu. Using the "No symmetry" option, only the figure that the user draws is shown. Using "2-way symmetry", the user's figure and its horizontal reflection are shown. Using "4-way symmetry", the two vertical reflections are added. Finally, using "8-way symmetry", the four diagonal reflections are also added. Formulas for computing the reflections are given below.

The source code [SimplePaint3.java](#) already has a `drawFigure()` subroutine that draws all the figures. You can add a `putMultiFigure()` routine to draw a figure and some or all of its reflections. `putMultiFigure` should call the existing `drawFigure` to draw the figure and any necessary reflections. It decides which reflections to draw based on the setting of the symmetry menu. Where the `mousePressed`, `mouseDragged`, and `mouseReleased` methods call `drawFigure`, they should call `putMultiFigure` instead. The source code also has a `repaintRect()` method that calls `repaint()` on a rectangle that contains two given points. You can treat this in the same way as `drawFigure()`, adding a `repaintMultiRect()` that calls `repaintRect()` and replacing each call to `repaintRect()` with a call to `repaintMultiRect()`. Alternatively, if you are willing to let your applet be a little less efficient about repainting, you could simply replace each call to `repaintRect()` with a simple call to `repaint()`, without parameters. This just means that the applet will redraw a larger area than it really needs to.

If (x, y) is a point in a component that is `width` pixels wide and `height` pixels high, then the reflections of this point are obtained as follows:

The horizontal reflection is $(width - x, y)$

The two vertical reflections are $(x, height - y)$ and $(width - x, height - y)$

To get the four diagonal reflections, first compute the diagonal reflection of (x, y) as

```
a = (int)((double)y / height) * width ;
b = (int)((double)x / width) * height ;
```

Then use the horizontal and vertical reflections of the point (a, b) :

```
(a, b)
(width - a, b)
```

```
(a, height - b)
(width - a, height - b)
```

(The diagonal reflections are harder than they would be if the canvas were square. Then the height would equal the width, and the reflection of (x, y) would just be (y, x) .)

To reflect a figure determined by two points, (x_1, y_1) and (x_2, y_2) , compute the reflections of both points to get the reflected figure.

This is really not so hard. The changes you have to make to the source code are not as long as the explanation I have given here.

Here is my applet. Don't forget to try it with the symmetry menu set to "8-way Symmetry"!

[See the solution!](#)

Exercise 7.8: Turn your applet from the previous exercise into a stand-alone application that runs as a `JFrame`. (If you didn't do the previous exercise, you can do this exercise with the original [SimplePaint3.java](#).) To make the exercise more interesting, remove the `JButtons` and `JComboBoxes` and replace them with a menubar at the top of the frame. You can design the menus any way you like, but you should have at least the same functionality as in the original program.

As an improvement, you might add an "Undo" command. When the user clicks on the "Undo" button, the previous drawing operation will be undone. This just means returning to the image as it was before the drawing operation took place. This is easy to implement, as long as we allow just one operation to be undone. When the off-screen canvas, `OSI`, is created, make a second off-screen canvas, `undoBuffer`, of the same size. Before starting any drawing operation, copy the image from `OSI` to `undoBuffer`. You can do this with the commands

```
Graphics undoGr = undoBuffer.getGraphics();
undoGr.drawImage(OSI, 0, 0, null);
```

When the user clicks "Undo", just swap the values of `OSI` and `undoBuffer` and repaint. The previous image will appear on the screen. Clicking on "Undo" again will "undo the undo."

As another improvement, you could make it possible for the user to select a drawing color using a `JColorChooser` dialog box.

Here is a button that opens my program in its own window. (You don't have to write an applet to launch your frame. Just create the frame in the program's `main()` routine.)

[See the solution!](#)

[[Chapter Index](#) | [Main Index](#)]

Quiz Questions For Chapter 7

THIS PAGE CONTAINS A SAMPLE quiz on material from [Chapter 7](#) of this [on-line Java textbook](#). You should be able to answer these questions after studying that chapter. Sample answers to all the quiz questions can be found [here](#).

Question 1: What is the `FontMetrics` class used for?

Question 2: An *off-screen image* can be used to do *double buffering*. Explain this. (What are off-screen images? How are they used? Why are they important? What does this have to do with animation?)

Question 3: One of the main classes in Swing is the `JComponent` class. What is meant by a component? What are some examples?

Question 4: What is the function of a *LayoutManager* in Java?

Question 5: What does it mean to use a `null` layout manager, and why would you want to do so?

Question 6: What is a `JCheckBox` and how is it used?

Question 7: What is a *thread*

Question 8: Explain how `Timers` are used to do animation.

Question 9: Menus can contain *sub-menus*. What does this mean, and how are sub-menus handled in Java?

Question 10: What is the purpose of the `JFrame` class?

[[Answers](#) | [Chapter Index](#) | [Main Index](#)]

Chapter 8

Arrays

COMPUTERS GET A LOT OF THEIR POWER from working with **data structures**. A data structure is an organized collection of related data. An object is a data structure, but this type of data structure -- consisting of a fairly small number of named instance variables -- is just the beginning. In many cases, programmers build complicated data structures by hand, by linking objects together. We'll look at these custom-built data structures in Chapter 11. But there is one type of data structure that is so important and so basic that it is built into every programming language: the array.

An **array** is a data structure consisting of a numbered list of items, where all the items are of the same type. In Java, the items in an array are always numbered from zero up to some maximum value, which is set when the array is created. For example, an array might contain 100 integers, numbered from zero to 99. The items in an array can belong to one of Java's primitive types. They can also be references to objects, so that you could, for example, make an array containing all the components in an applet.

This chapter discusses how arrays are created and used in Java. It also covers the standard class `java.util.ArrayList`. An object of type `ArrayList` is very similar to an array of `Objects`, but it can grow to hold any number of items.

Contents of Chapter 8:

- Section 1: [Creating and Using Arrays](#)
 - Section 2: [Programming with Arrays](#)
 - Section 3: [Dynamic Arrays, ArrayLists, and Vectors](#)
 - Section 4: [Searching and Sorting](#)
 - Section 5: [Multi-Dimensional Arrays](#)
 - [Programming Exercises](#)
 - [Quiz on this Chapter](#)
-

[[First Section](#) | [Next Chapter](#) | [Previous Chapter](#) | [Main Index](#)]

Section 8.1

Creating and Using Arrays

WHEN A NUMBER OF DATA ITEMS are chunked together into a unit, the result is a **data structure**. Data structures can have very complex structure, but in many applications, the appropriate data structure consists simply of a sequence of data items. Data structures of this simple variety can be either **arrays** or **records**.

The term "record" is not used in Java. A record is essentially the same as a Java object that has instance variables only, but no instance methods. Some other languages, which do not support objects in general, nevertheless do support records. The C programming language, for example, is not object-oriented, but it has records, which in C go by the name "struct." The data items in a record -- in Java, an object's instance variables -- are called the **fields** of the record. Each item is referred to using a **field name**. In Java, field names are just the names of the instance variables. The distinguishing characteristics of a record are that the data items in the record are referred to by name and that different fields in a record are allowed to be of different types. For example, if the class `Person` is defined as:

```
class Person {
    String name;
    int id_number;
    Date birthday;
    int age;
}
```

then an object of class `Person` could be considered to be a record with four fields. The field names are `name`, `id_number`, `birthday`, and `age`. Note that the fields are of various types: `String`, `int`, and `Date`.

Because records are just a special type of object, I will not discuss them further.

Like a record, an array is a sequence of items. However, where items in a record are referred to by **name**, the items in an array are numbered, and individual items are referred to by their **position number**. Furthermore, all the items in an array must be of the same type. The definition of an array is: a numbered sequence of items, which are all of the same type. The number of items in an array is called the **length** of the array. The position number of an item in an array is called the **index** of that item. The type of the individual items in an array is called the **base type** of the array.

The base type of an array can be any Java type, that is, one of the primitive types, or a class name, or an interface name. If the base type of an array is `int`, it is referred to as an "array of `ints`." An array with base type `String` is referred to as an "array of `Strings`." However, an array is not, properly speaking, a list of integers or strings or other values. It is better thought of as a list of variables of type `int`, or of type `String`, or of some other type. As always, there is some potential for confusion between the two uses of a variable: as a name for a memory location and as a name for the value stored in that memory location. Each position in an array acts as a variable. Each position can hold a value of a specified type (the base type of the array). The value can be changed at any time. Values are stored in an array. The array is the container, not the values.

The items in an array -- really, the individual variables that make up the array -- are more often referred to as the **elements** of the array. In Java, the elements in an array are always numbered starting from zero. That is, the index of the first element in the array is zero. If the length of the array is N , then the index of the last element in the array is $N-1$. Once an array has been created, its length cannot be changed.

Java arrays are objects. This has several consequences. Arrays are created using a form of the `new`

operator. No variable can ever hold an array; a variable can only refer to an array. Any variable that can refer to an array can also hold the value `null`, meaning that it doesn't at the moment refer to anything. Like any object, an array belongs to a class, which like all classes is a subclass of the class `Object`. The elements of the array are, essentially, instance variables in the array object, except that they are referred to by number rather than by name.

Nevertheless, even though arrays are objects, there are differences between arrays and other kinds of objects, and there are a number of special language features in Java for creating and using arrays.

Suppose that `A` is a variable that refers to an array. Then the item at index `k` in `A` is referred to as `A[k]`. The first item is `A[0]`, the second is `A[1]`, and so forth. "`A[k]`" is really a variable, and it can be used just like any other variable. You can assign values to it, you can use it in expressions, and you can pass it as a parameter to subroutines. All of this will be discussed in more detail below. For now, just keep in mind the syntax

array-variable [integer-expression]

for referring to an element of an array.

Although every array, as an object, is a member of some class, array classes never have to be defined. Once a type exists, the corresponding array class exists automatically. If the name of the type is `BaseType`, then the name of the associated array class is `BaseType[]`. That is to say, an object belonging to the class `BaseType[]` is an array of items, where each item is a variable of type `BaseType`. The brackets, "`[]`", are meant to recall the syntax for referring to the individual items in the array. "`BaseType[]`" is read as "array of `BaseType`" or "`BaseType` array." It might be worth mentioning here that if `ClassA` is a subclass of `ClassB`, then `ClassA[]` is automatically a subclass of `ClassB[]`.

The base type of an array can be any legal Java type. From the primitive type `int`, the array type `int[]` is derived. Each element in an array of type `int[]` is a variable of type `int`, which holds a value of type `int`. From a class named `Shape`, the array type `Shape[]` is derived. Each item in an array of type `Shape[]` is a variable of type `Shape`, which holds a value of type `Shape`. This value can be either `null` or a reference to an object belonging to the class `Shape`. (This includes objects belonging to subclasses of `Shape`.)

Let's try to get a little more concrete about all this, using arrays of integers as our first example. Since `int[]` is a class, it can be used to declare variables. For example,

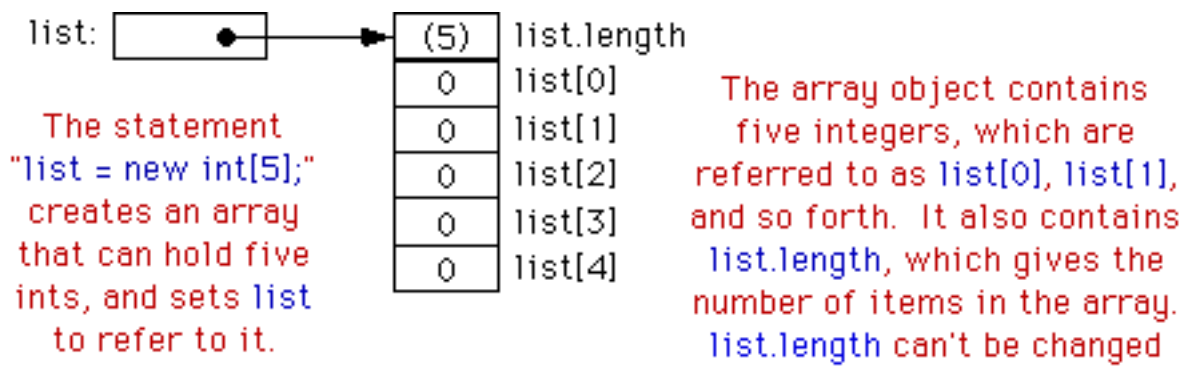
```
int[] list;
```

creates a variable named `list` of type `int[]`. This variable is capable of referring to an array of `ints`, but initially its value is `null` (if it is a member variable in a class) or undefined (if it is a local variable in a method). The `new` operator is used to create a new array object, which can then be assigned to `list`. The syntax for using `new` with arrays is different from the syntax you learned previously. As an example,

```
list = new int[5];
```

creates an array of five integers. More generally, the constructor "`new BaseType[N]`" is used to create an array belonging to the class `BaseType[]`. The value `N` in brackets specifies the length of the array, that is, the number of elements that it contains. Note that the array "knows" how long it is. The length of the array is an instance variable in the array object. In fact, the length of an array, `list`, can be referred to as `list.length`. (However, you are not allowed to change the value of `list.length`, so it's really a "final" instance variable, that is, one whose value cannot be changed after it has been initialized.)

The situation produced by the statement "`list = new int[5];`" can be pictured like this:



Note that the newly created array of integers is automatically filled with zeros. In Java, a newly created array is always filled with a known, default value: zero for numbers, false for boolean, the character with Unicode number zero for char, and null for objects.

The elements in the array, `list`, are referred to as `list[0]`, `list[1]`, `list[2]`, `list[3]`, and `list[4]`. (Note again that the index for the last item is one less than `list.length`.) However, array references can be much more general than this. The brackets in an array reference can contain any expression whose value is an integer. For example if `indx` is a variable of type `int`, then `list[indx]` and `list[2*indx+7]` are syntactically correct references to elements of the array `list`. Thus, the following loop would print all the integers in the array, `list`, to standard output:

```
for (int i = 0; i < list.length; i++) {
    System.out.println( list[i] );
}
```

The first time through the loop, `i` is 0, and `list[i]` refers to `list[0]`. So, it is the value stored in the variable `list[0]` that is printed. The second time through the loop, `i` is 1, and the value stored in `list[1]` is printed. The loop ends after printing the value of `list[4]`, when `i` becomes equal to 5 and the continuation condition "`i < list.length`" is no longer true. This is a typical example of using a loop to process an array. I'll discuss more examples of array processing throughout this chapter.

Every use of a variable in a program specifies a memory location. Think for a moment about what the computer does when it encounters a reference to an array element, `list[k]`, while it is executing a program. The computer must determine which memory location is being referred to. To the computer, `list[k]` means something like this: "Get the pointer that is stored in the variable, `list`. Follow this pointer to find an array object. Get the value of `k`. Go to the `k`-th position in the array, and that's the memory location you want." There are two things that can go wrong here. Suppose that the value of `list` is null. If that is the case, then `list` doesn't even refer to an array. The attempt to refer to an element of an array that doesn't exist is an error. This is an example of a "null pointer" error. The second possible error occurs if `list` does refer to an array, but the value of `k` is outside the legal range of indices for that array. This will happen if `k < 0` or if `k >= list.length`. This is called an "array index out of bounds" error. When you use arrays in a program, you should be mindful that both types of errors are possible. However, array index out of bounds errors are by far the most common error when working with arrays.

For an array variable, just as for any variable, you can declare the variable and initialize it in a single step. For example,

```
int[] list = new int[5];
```

If `list` is a local variable in a subroutine, then this is exactly equivalent to the two statements:

```
int[] list;
list = new int[5];
```

(If `list` is an instance variable, then of course you can't simply replace `int[] list = new`

`int[5];` with `int[] list; list = new int[5];` since the assignment statement `list = new int[5];` is only legal inside a subroutine.)

The new array is filled with the default value appropriate for the base type of the array -- zero for `int` and `null` for class types, for example. However, Java also provides a way to initialize an array variable with a new array filled with a specified list of values. In a declaration statement that creates a new array, this is done with an **array initializer**. For example,

```
int[] list = { 1, 4, 9, 16, 25, 36, 49 };
```

creates a new array containing the seven values 1, 4, 9, 16, 25, 36, and 49, and sets `list` to refer to that new array. The value of `list[0]` will be 1, the value of `list[1]` will be 4, and so forth. The length of `list` is seven, since seven values are provided in the initializer.

An array initializer takes the form of a list of values, separated by commas and enclosed between braces. The length of the array does not have to be specified, because it is implicit in the list of values. The items in an array initializer don't have to be constants. They can be variables or arbitrary expressions, provided that their values are of the appropriate type. For example, the following declaration creates an array of eight `Colors`. Some of the colors are given by expressions of the form `"new Color(r,g,b)"`:

```
Color[] palette =
{
    Color.black,
    Color.red,
    Color.pink,
    new Color(0,180,0), // dark green
    Color.green,
    Color.blue,
    new Color(180,180,255), // light blue
    Color.white
};
```

A list initializer of this form can be used only in a declaration statement, to give an initial value to a newly declared array variable. It cannot be used in an assignment statement to assign a value to a variable that has been previously declared. However, there is another, similar notation for creating a new array that can be used in an assignment statement or passed as a parameter to a subroutine. The notation uses another form of the `new` operator to create and initialize a new array object. (This rather odd syntax is reminiscent of the syntax for anonymous classes, which were discussed in [Section 5.6](#).) For example to assign a new value to an array variable, `list`, that was declared previously, you could use:

```
list = new int[] { 1, 8, 27, 64, 125, 216, 343 };
```

The general syntax for this form of the `new` operator is

```
new base-type [ ] { list-of-values }
```

This is an expression whose value is an array object. It can be used in any context where an object of type `base-type[]` is expected. For example, if `makeButtons` is a method that takes an array of `Strings` as a parameter, you could say:

```
makeButtons( new String[] { "Stop", "Go", "Next", "Previous" } );
```

One final note: For historical reasons, the declaration

```
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.

[[Next Section](#) | [Previous Chapter](#) | [Chapter Index](#) | [Main Index](#)]

Section 8.2

Programming with Arrays

ARRAYS ARE THE MOST BASIC AND THE MOST IMPORTANT type of data structure, and techniques for processing arrays are among the most important programming techniques you can learn. Two fundamental array processing techniques -- searching and sorting -- will be covered in [Section 4](#). This section introduces some of the basic ideas of array processing in general.

In many cases, processing an array means applying the same operation to each item in the array. This is commonly done with a `for` loop. A loop for processing all the items in an array `A` has the form:

```
// do any necessary initialization
for (int i = 0; i < A.length; i++) {
    . . . // process A[i]
}
```

Suppose, for example, that `A` is an array of type `double[]`. Suppose that the goal is to add up all the numbers in the array. An informal algorithm for doing this would be:

```
Start with 0;
Add A[0];    (process the first item in A)
Add A[1];    (process the second item in A)
.
.
.
Add A[ A.length - 1 ];    (process the last item in A)
```

Putting the obvious repetition into a loop and giving a name to the sum, this becomes:

```
double sum; // The sum of the numbers in A.
sum = 0;    // Start with 0.
for (int i = 0; i < A.length; i++)
    sum += A[i]; // add A[i] to the sum, for
                // i = 0, 1, ..., A.length - 1
```

Note that the continuation condition, "`i < A.length`", implies that the last value of `i` that is actually processed is `A.length-1`, which is the index of the final item in the array. It's important to use "`<`" here, not "`<=`", since "`<=`" would give an array index out of bounds error.

Eventually, you should just about be able to write loops similar to this one in your sleep. I will give a few more simple examples. Here is a loop that will count the number of items in the array `A` which are less than zero:

```
int count; // For counting the items.
count = 0; // Start with 0 items counted.
for (int i = 0; i < A.length; i++) {
    if (A[i] < 0.0) // if this item is less than zero...
        count++; // ...then count it
}
// At this point, the value of count is the number
// of items that have passed the test of being < 0
```

Replace the test "`A[i] < 0.0`", if you want to count the number of items in an array that satisfy some other property. Here is a variation on the same theme. Suppose you want to count the number of times that an item in the array `A` is equal to the item that follows it. The item that follows `A[i]` in the array is `A[i+1]`, so the test in this case is "`if (A[i] == A[i+1])`". But there is a catch: This test cannot be

applied when `A[i]` is the last item in the array, since then there is no such item as `A[i+1]`. The result of trying to apply the test in this case would be an array index out of bounds error. This just means that we have to stop one item short of the final item:

```
int count = 0;
for (int i = 0; i < A.length - 1; i++) {
    if (A[i] == A[i+1])
        count++;
}
```

Another typical problem is to find the largest number in `A`. The strategy is to go through the array, keeping track of the largest number found so far. We'll store the largest number found so far in a variable called `max`. As we look through the array, whenever we find a number larger than the current value of `max`, we change the value of `max` to that larger value. After the whole array has been processed, `max` is the largest item in the array overall. The only question is, what should the original value of `max` be? One possibility is to start with `max` equal to `A[0]`, and then to look through the rest of the array, starting from `A[1]`, for larger items:

```
double max = A[0];
for (int i = 1; i < A.length; i++) {
    if (A[i] > max)
        max = A[i];
}
// at this point, max is the largest item in A
```

(There is one subtle problem here. It's possible in Java for an array to have length zero. In that case, `A[0]` doesn't exist, and the reference to `A[0]` in the first line gives an array index out of bounds error. However, zero-length arrays are normally something that you want to avoid in real problems. Anyway, what would it mean to ask for the largest item in an array that contains no items at all?)

As a final example of basic array operations, consider the problem of copying an array. To make a copy of our sample array `A`, it is **not sufficient to say**

```
double[] B = A;
```

since this does not create a new array object. All it does is declare a new array variable and make it refer to the same object to which `A` refers. (So that, for example, a change to `A[i]` will automatically change `B[i]` as well.) To make a new array that is a copy of `A`, it is necessary to make a new array object and to copy each of the individual items from `A` into the new array:

```
double[] B = new double[A.length]; // Make a new array object,
                                   // the same size as A.
for (int i = 0; i < A.length; i++)
    B[i] = A[i]; // Copy each item from A to B.
```

Copying values from one array to another is such a common operation that Java has a predefined subroutine to do it. The subroutine, `System.arraycopy()`, is a static member subroutine in the standard `System` class. Its declaration has the form

```
public static void arraycopy(Object sourceArray, int sourceStartIndex,
                             Object destArray, int destStartIndex, int count)
```

where `sourceArray` and `destArray` can be arrays with any base type. Values are copied from `sourceArray` to `destArray`. The `count` tells how many elements to copy. Values are taken from `sourceArray` starting at position `sourceStartIndex` and are stored in `destArray` starting at position `destStartIndex`. For example, to make a copy of the array, `A`, using this subroutine, you would say:

```
double B = new double[A.length];
System.arraycopy( A, 0, B, 0, A.length );
```

An array type, such as `double[]`, is a full-fledged Java type, so it can be used in all the ways that any other Java type can be used. In particular, it can be used as the type of a formal parameter in a subroutine. It can even be the return type of a function. For example, it might be useful to have a function that makes a copy of an array of doubles:

```
double[] copy( double[] source ) {
    // Create and return a copy of the array, source.
    // If source is null, return null.
    if ( source == null )
        return null;
    double[] cpy; // A copy of the source array.
    cpy = new double[source.length];
    System.arraycopy( source, 0, cpy, 0, source.length );
    return cpy;
}
```

The `main()` routine of a program has a parameter of type `String[]`. You've seen this used since all the way back in [Chapter 2](#), but I haven't really been able to explain it until now. The parameter to the `main()` routine is an array of `Strings`. When the system calls the `main()` routine, the strings in this array are the **command-line parameters**. 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 parameters. For example, if the name of the class that contains the `main()` routine is `myProg`, then the user can type `"java myProg"` to execute the program. In this case, there are no command-line parameters. But if the user types the command `"java myProg one two three"`, then the command-line parameters 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 parameters entered by the user:

```
public class CLDemo {

    public static void main(String[] args) {
        System.out.println("You entered " + args.length
                           + " command-line parameters.");

        if (args.length > 0) {
            System.out.println("They were:");
            for (int i = 0; i < args.length; i++)
                System.out.println("    " + args[i]);
        }
    } // end main()

} // end class CLDemo
```

Note that the parameter, `args`, is never `null` when `main()` is called by the system, but it might be an array of length zero.

In practice, command-line parameters are often the names of files to be processed by the program. I will give some examples of this in [Chapter 10](#), when I discuss file processing.

So far, all my examples of array processing have used **sequential access**. That is, the elements of the array were processed one after the other in the sequence in which they occur in the array. But one of the big advantages of arrays is that they allow **random access**. That is, every element of the array is equally accessible at any given time.

As an example, let's look at a well-known problem called the birthday problem: Suppose that there are N people in a room. What's the chance that there are two people in the room who have the same birthday? (That is, they were born on the same day in the same month, but not necessarily in the same year.) Most people severely underestimate the probability. We actually look at a different version of the problem: 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 know, has this birthday already been used? The answer is a boolean value, true or false. To hold this data, we can use an array of 365 boolean values:

```
boolean[] used;
used = new boolean[365];
```

The days of the year are numbered from 0 to 364. The value of `used[i]` is true if someone has been selected whose birthday is day number i . Initially, all the values in the array, `used`, are false. When we select someone whose birthday is day number i , we first check whether `used[i]` is true. If so, then this is the second person with that birthday. We are done. If `used[i]` is false, we set `used[i]` to be true to record the fact that we've encountered someone with that birthday, and we go on to the next person. Here is a subroutine that carries out the simulated experiment (Of course, in the subroutine, there are no simulated people, only simulated birthdays):

```
static void birthdayProblem() {
    // 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.

    boolean[] used; // For recording the possible birthdays
                   // that have been seen so far.  A value
                   // of true in used[i] means that a person
                   // whose birthday is the i-th day of the
                   // year has been found.

    int count;      // The number of people who have been checked.

    used = new boolean[365]; // Initially, all entries are false.

    count = 0;

    while (true) {
        // Select a birthday at random, from 0 to 364.
        // If the birthday has already been used, quit.
        // Otherwise, record the birthday as used.
        int birthday; // The selected birthday.
        birthday = (int)(Math.random()*365);
        count++;
        if ( used[birthday] )
            break;
        used[birthday] = true;
    }

    TextIO.putln("A duplicate birthday was found after "
```

```
+ count + " tries.");
```

```
} // end birthdayProblem()
```

This subroutine makes essential use of the fact that every element in a newly created array of `booleans` is set to be `false`. If we wanted to reuse the same array in a second simulation, we would have to reset all the elements in it to be `false` with a `for` loop

```
for (int i = 0; i < 365; i++)
    used[i] = false;
```

Here is an applet that will run the simulation as many times as you like. Are you surprised at how few people have to be chosen, in general?

(Applet "BirthdayProblemConsole" would be displayed here
if Java were available.)

One of the examples in [Section 6.4](#) was an applet that shows multiple copies of a message in random positions, colors, and fonts. When the user clicks on the applet, the positions, colors, and fonts are changed to new random values. Like several other examples from that chapter, the applet had a flaw: It didn't have any way of storing the data that would be necessary to redraw itself. Chapter 7 introduced off-screen canvases as a solution to this problem, but off-screen canvases are not a good solution in every case. Arrays provide us with an alternative solution. Here's a new version of the applet. This version uses an array to store the position, font, and color of each string. When the applet is painted, this information is used to draw the strings, so it will redraw itself correctly when it is covered and then uncovered. When you click on the applet, the array is filled with new random values and the applet is repainted.

(Applet "RandomStringsWithArray" would be displayed here
if Java were available.)

In this applet, the number of copies of the message is given by a named constant, `MESSAGE_COUNT`. One way to store the position, color, and font of `MESSAGE_COUNT` strings would be to use four arrays:

```
int[] x = new int[MESSAGE_COUNT];
int[] y = new int[MESSAGE_COUNT];
Color[] color = new Color[MESSAGE_COUNT];
Font[] font = new Font[MESSAGE_COUNT];
```

These arrays would be filled with random values. In the `paintComponent()` method, the *i*-th copy of the string would be drawn at the point `(x[i],y[i])`. Its color would be given by `color[i]`. And it would be drawn in the font `font[i]`. This would be accomplished by the `paintComponent()` method

```
public void paintComponent(Graphics g) {
    super.paintComponent(); // (Fill with background color.)
    for (int i = 0; i < MESSAGE_COUNT; i++) {
        g.setColor( color[i] );
        g.setFont( font[i] );
        g.drawString( message, x[i], y[i] );
    }
}
```

This approach is said to use **parallel arrays**. The data for a given copy of the message is spread out across several arrays. If you think of the arrays as laid out in parallel columns -- array `x` in the first column, array `y` in the second, array `color` in the third, and array `font` in the fourth -- then the data for the *i*-th string can be found along the *i*-th row. There is nothing wrong with using parallel arrays in this simple example, but it does go against the object-oriented philosophy of keeping related data in one object. If we

follow this rule, then we don't have to imagine the relationship among the data because all the data for one copy of the message is physically in one place. So, when I wrote the applet, I made a simple class to represent all the data that is needed for one copy of message:

```
class StringData {
    // Data for one copy of the message.
    int x,y;           // Position of the message.
    Color color;       // Color of the message.
    Font font;         // Font used for the message.
}
```

(This class is actually defined as a static nested class in the main applet class.) To store the data for multiple copies of the message, I use an array of type `StringData[]`. The array is declared as an instance variable, with the name `data`:

```
StringData[] data;
```

Of course, the value of `data` is `null` until an actual array is created and assigned to it. This is done in the `init()` method of the applet with the statement

```
data = new StringData[MESSAGE_COUNT];
```

Just after this array is created, the value of each element in the array is `null`. We want to store data in objects of type `StringData`, but no such objects exist yet. All we have is an array of variables that are capable of referring to such objects. I decided to create the objects in the applet's `init` method. (It could be done in other places -- just so long as we avoid trying to use to an object that doesn't exist. This is important: Remember that a newly created array whose base type is an object type is always filled with `null` elements. There are *no* objects in the array until you put them there.) The objects are created with the `for` loop

```
for (int i = 0; i < MESSAGE_COUNT; i++)
    data[i] = new StringData();
```

Now, the idea is to store data for the *i*-th copy of the message in the variables `data[i].x`, `data[i].y`, `data[i].color`, and `data[i].font`. (Make sure that you understand the notation here: `data[i]` refers to an object. That object contains instance variables. The notation `data[i].x` tells the computer: "Find your way to the object that is referred to by `data[i]`. Then go to the instance variable named `x` in that object." Variable names can get even more complicated than this.) Using the array, `data`, the `paintComponent()` method for the applet becomes

```
public void paintComponent(Graphics g) {
    super.paintComponent(g); // (Fill with background color.)
    for (int i = 0; i < MESSAGE_COUNT; i++) {
        g.setColor( data[i].color );
        g.setFont( data[i].font );
        g.drawString( message, data[i].x, data[i].y );
    }
}
```

There is still the matter of filling the array, `data`, with random values. If you are interested, you can look at the source code for the applet, [RandomStringsWithArray.java](#).

The applet actually uses one other array. The font for a given copy of the message is chosen at random from a set of five possible fonts. In the original version of the applet, there were five variables of type `Font` to represent the fonts. The variables were named `font1`, `font2`, `font3`, `font4`, and `font5`. To select one of these fonts at random, a `switch` statement could be used:

```
Font randomFont; // One of the 5 fonts, chosen at random.
int rand;        // A random integer in the range 0 to 4.
```

```

rand = (int)(Math.random() * 5);
switch (rand) {
    case 0:
        randomFont = font1;
        break;
    case 1:
        randomFont = font2;
        break;
    case 2:
        randomFont = font3;
        break;
    case 3:
        randomFont = font4;
        break;
    case 4:
        randomFont = font5;
        break;
}

```

In the new version of the applet, the five fonts are stored in an array, which is named `fonts`. This array is declared as an instance variable

```
Font[] fonts;
```

The array is created and filled with fonts in the `init()` method:

```

fonts = new Font[5]; // Array to store five fonts.
fonts[0] = new Font("Serif", Font.BOLD, 14);
fonts[1] = new Font("SansSerif", Font.BOLD + Font.ITALIC, 24);
fonts[2] = new Font("Monospaced", Font.PLAIN, 20);
fonts[3] = new Font("Dialog", Font.PLAIN, 30);
fonts[4] = new Font("Serif", Font.ITALIC, 36);

```

This makes it much easier to select one of the fonts at random. It can be done with the statements

```

Font randomFont; // One of the 5 fonts, chosen at random.
int fontIndex;   // A random number in the range 0 to 4.
fontIndex = (int)(Math.random() * 5);
randomFont = fonts[ fontIndex ];

```

The `switch` statement has been replaced by a single line of code. This is a very typical application of arrays. Here is another example of the same sort of thing. Months are often stored as numbers 1, 2, 3, ..., 12. Sometimes, however, these numbers have to be translated into the names January, February, ..., December. The translation can be done with an array. The array could be declared and initialized as

```

static String[] monthName = { "January", "February", "March",
                              "April",   "May",       "June",
                              "July",    "August",   "September",
                              "October", "November", "December" };

```

If `mth` is a variable that holds one of the integers 1 through 12, then `monthName[mth-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!

Section 8.3

Dynamic Arrays, ArrayLists, and Vectors

THE SIZE OF AN ARRAY is fixed when it is created. In many cases, however, the number of data items that are actually stored in the array varies with time. Consider the following examples: An array that stores the lines of text in a word-processing program. An array that holds the list of computers that are currently downloading a page from a Web site. An array that contains the shapes that have been added to the screen by the user of a drawing program. Clearly, we need some way to deal with cases where the number of data items in an array is not fixed.

Partially Full Arrays

Consider an application where the number of items that we want to store in an array changes as the program runs. Since the size of the array can't actually be changed, a separate counter variable must be used to keep track of how many spaces in the array are in use. (Of course, every space in the array has to contain something; the question is, how many spaces contain useful or valid items?)

Consider, for example, a program that reads positive integers entered by the user and stores them for later processing. The program stops reading when the user inputs a number that is less than or equal to zero. The input numbers can be kept in an array, `numbers`, of type `int[]`. Let's say that no more than 100 numbers will be input. Then the size of the array can be fixed at 100. But the program must keep track of how many numbers have actually been read and stored in the array. For this, it can use an integer variable, `numCt`. Each time a number is stored in the array, `numCt` must be incremented by one. As a rather silly example, let's write a program that will read the numbers input by the user and then print them in reverse order. (This is, at least, a processing task that requires that the numbers be saved in an array. Remember that many types of processing, such as finding the sum or average or maximum of the numbers, can be done without saving the individual numbers.)

```
public class ReverseInputNumbers {

    public static void main(String[] args) {

        int[] numbers; // An array for storing the input values.
        int numCt;      // The number of numbers saved in the array.
        int num;        // One of the numbers input by the user.

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

        TextIO.putln("Enter up to 100 positive integers; enter 0 to end.");

        while (true) { // Get the numbers and put them in the array.
            TextIO.put("? ");
            num = TextIO.getlnInt();
            if (num <= 0)
                break;
            numbers[numCt] = num;
            numCt++;
        }

        TextIO.putln("\nYour numbers in reverse order are:\n");
    }
}
```

```

        for (int i = numCt - 1; i >= 0; i--) {
            TextIO.putln( numbers[i] );
        }

    } // end main();

} // end class ReverseInputNumbers

```

It is especially important to note that the variable `numCt` plays a dual role. It is the number of numbers that have been entered into the array. But it is also the index of the next available spot in the array. For example, if 4 numbers have been stored in the array, they occupy locations number 0, 1, 2, and 3. The next available spot is location 4. When the time comes to print out the numbers in the array, the last occupied spot in the array is location `numCt - 1`, so the `for` loop prints out values starting from location `numCt - 1` and going down to 0.

Let's look at another, more realistic example. Suppose that you write a game program, and that players can join the game and leave the game as it progresses. As a good object-oriented programmer, you probably have a class named `Player` to represent the individual players in the game. A list of all players who are currently in the game could be stored in an array, `playerList`, of type `Player[]`. Since the number of players can change, you will also need a variable, `playerCt`, to record the number of players currently in the game. Assuming that there will never be more than 10 players in the game, you could declare the variables as:

```

Player[] playerList = new Player[10]; // Up to 10 players.
int      playerCt   = 0; // At the start, there are no players.

```

After some players have joined the game, `playerCt` will be greater than 0, and the player objects representing the players will be stored in the array elements `playerList[0]`, `playerList[1]`, ..., `playerList[playerCt-1]`. Note that the array element `playerList[playerCt]` is **not in use**. **The procedure for adding a new player, `newPlayer`, to the game is simple:**

```

playerList[playerCt] = newPlayer; // Put new player in next
                                // available spot.
playerCt++; // And increment playerCt to count the new player.

```

Deleting a player from the game is a little harder, since you don't want to leave a "hole" in the array. Suppose you want to delete the player at index `k` in `playerList`. If you are not worried about keeping the players in any particular order, then one way to do this is to move the player from the last occupied position in the array into position `k` and then to decrement the value of `playerCt`:

```

playerList[k] = playerList[playerCt - 1];
playerCt--;

```

The player previously in position `k` is no longer in the array. The player previously in position `playerCt - 1` is now in the array twice. But it's only in the occupied or valid part of the array once, since `playerCt` has decreased by one. Remember that every element of the array has to hold some value, but only the values in positions 0 through `playerCt - 1` will be looked at or processed in any way.

Suppose that when deleting the player in position `k`, you'd like to keep the remaining players in the same order. (Maybe because they take turns in the order in which they are stored in the array.) To do this, all the players in positions `k+1` and above must move down one position in the array. Player `k+1` replaces player `k`, who is out of the game. Player `k+2` fills the spot left open when player `k+1` moved. And so on. The code for this is

```

for (int i = k+1; i < playerCt; i++) {
    playerList[i-1] = playerList[i];
}

```



```
playerCt--;
```

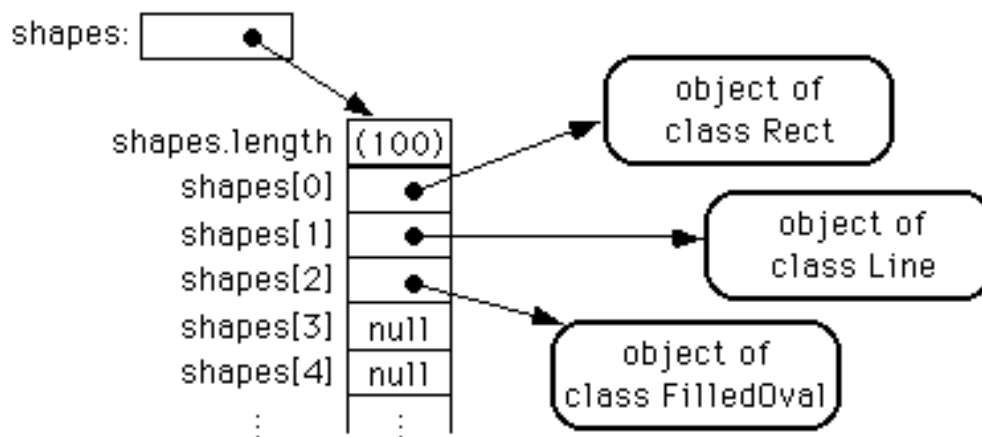
It's worth emphasizing that the `Player` example deals with an array whose base type is a class. An item in the array is either `null` or is a reference to an object belonging to the class, `Player`. The `Player` objects themselves are not really stored in the array, only references to them. Note that because of the rules for assignment in Java, the objects can actually belong to subclasses of `Player`. Thus there could be different classes of `Players` such as computer players, regular human players, players who are wizards, ..., all represented by different subclasses of `Player`.

As another example, suppose that a class `Shape` represents the general idea of a shape drawn on a screen, and that it has subclasses to represent specific types of shapes such as lines, rectangles, rounded rectangles, ovals, filled-in ovals, and so forth. (`Shape` itself would be an abstract class, as discussed in [Section 5.4](#).)

Then an array of type `Shape[]` can hold references to objects belonging to the subclasses of `Shape`. For example, the situation created by the statements

```
Shape[] shapes = new Shape[100]; // Array to hold up to 100 shapes.
shapes[0] = new Rect();           // Put some objects in the array.
shapes[1] = new Line();           //      (A real program would
shapes[2] = new FilledOval();     //      use some parameters here.)
int shapeCt = 3; // Keep track of number of objects in array.
```

could be illustrated as:



Such an array would be useful in a drawing program. The array could be used to hold a list of shapes to be displayed. If the `Shape` class includes a method, `"void redraw(Graphics g)"` for drawing the shape in a graphics context `g`, then all the shapes in the array could be redrawn with a simple for loop:

```
for (int i = 0; i < shapeCt; i++)
    shapes[i].redraw(g);
```

The statement `"shapes[i].redraw(g);"` calls the `redraw()` method belonging to the particular shape at index `i` in the array. Each object knows how to redraw itself, so that repeated executions of the statement can produce a variety of different shapes on the screen. This is nice example both of polymorphism and of array processing.

Dynamic Arrays

In each of the above examples, an arbitrary limit was set on the number of items -- 100 ints, 10 `Players`, 100 `Shapes`. Since the size of an array is fixed, a given array can only hold a certain maximum number of items. In many cases, such an arbitrary limit is undesirable. Why should a program work for 100 data values, but not for 101? The obvious alternative of making an array that's so big that it will work in any practical

case is not usually a good solution to the problem. It means that in most cases, a lot of computer memory will be wasted on unused space in the array. That memory might be better used for something else. And what if someone is using a computer that could handle as many data values as the user actually wants to process, but doesn't have enough memory to accommodate all the extra space that you've allocated?

Clearly, it would be nice if we could increase the size of an array at will. This is not possible, but what is possible is just as good. Remember that an array variable does not actually hold an array. It just holds a reference to an array object. We can't make the array bigger, but we can make a new, bigger array object and change the value of the array variable so that it refers to the bigger array. Of course, we also have to copy the contents of the old array into the new array. The array variable then refers to an array object that contains all the data of the old array, with room for additional data. The old array will be garbage collected, since it is no longer in use.

Let's look back at the game example, in which `playerList` is an array of type `Player[]` and `playerCt` is the number of spaces that have been used in the array. Suppose that we don't want to put a pre-set limit on the number of players. If a new player joins the game and the current array is full, we just make a new, bigger one. The same variable, `playerList`, will refer to the new array. Note that after this is done, `playerList[0]` will refer to a different memory location, but the value stored in `playerList[0]` will still be the same as it was before. Here is some code that will do this:

```
// Add a new player, even if the current array is full.

if (playerCt == playerList.length) {
    // Array is full. Make a new, bigger array,
    // copy the contents of the old array into it,
    // and set playerList to refer to the new array.
    int newSize = 2 * playerList.length; // Size of new array.
    Player[] temp = new Player[newSize]; // The new array.
    System.arraycopy(playerList, 0, temp, 0, playerList.length);
    playerList = temp; // Set playerList to refer to new array.
}

// At this point, we KNOW there is room in the array.

playerList[playerCt] = newPlayer; // Add the new player...
playerCt++;                       // ...and count it.
```

If we are going to be doing things like this regularly, it would be nice to define a reusable class to handle the details. An array-like object that changes size to accommodate the amount of data that it actually contains is called a **dynamic array**. A dynamic array supports the same operations as an array: putting a value at a given position and getting the value that is stored at a given position. But there is no upper limit on the positions that can be used (except those imposed by the size of the computer's memory). In a dynamic array class, the `put` and `get` operations must be implemented as instance methods. Here, for example, is a class that implements a dynamic array of `ints`:

```
public class DynamicArrayOfInt {

    private int[] data; // An array to hold the data.

    public DynamicArrayOfInt() {
        // Constructor.
        data = new int[1]; // Array will grow as necessary.
    }

    public int get(int position) {
        // Get the value from the specified position in the array.
        // Since all array positions are initially zero, when the
```

```

        // specified position lies outside the actual physical size
        // of the data array, a value of 0 is returned.
        if (position >= data.length)
            return 0;
        else
            return data[position];
    }

    public void put(int position, int value) {
        // Store the value in the specified position in the array.
        // The data array will increase in size to include this
        // position, if necessary.
        if (position >= data.length) {
            // The specified position is outside the actual size of
            // the data array. Double the size, or if that still does
            // not include the specified position, set the new size
            // to 2*position.
            int newSize = 2 * data.length;
            if (position >= newSize)
                newSize = 2 * position;
            int[] newData = new int[newSize];
            System.arraycopy(data, 0, newData, 0, data.length);
            data = newData;
            // The following line is for demonstration purposes only.
            System.out.println("Size of dynamic array increased to "
                               + newSize);
        }
        data[position] = value;
    }
} // end class DynamicArrayOfInt

```

The data in a `DynamicArrayOfInt` object is actually stored in a regular array, but that array is discarded and replaced by a bigger array whenever necessary. If `numbers` is a variable of type `DynamicArrayOfInt`, then the command `numbers.put(pos, val)` stores the value `val` at position number `pos` in the dynamic array. The function `numbers.get(pos)` returns the value stored at position number `pos`.

The first example in this section used an array to store positive integers input by the user. We can rewrite that example to use a `DynamicArrayOfInt`. A reference to `numbers[i]` is replaced by `numbers.get(i)`. The statement `"numbers[numCt] = num;"` is replaced by `"numbers.put(numCt, num);"`. Here's the program:

```

public class ReverseWithDynamicArray {

    public static void main(String[] args) {

        DynamicArrayOfInt numbers; // To hold the input numbers.
        int numCt; // The number of numbers stored in the array.
        int num; // One of the numbers input by the user.

        numbers = new DynamicArrayOfInt();
        numCt = 0;

        TextIO.putln("Enter some positive integers; Enter 0 to end");
    }
}

```

```

        while (true) { // Get numbers and put them in the dynamic array.
            TextIO.put(" ? ");
            num = TextIO.getlnInt();
            if (num <= 0)
                break;
            numbers.put(numCt, num); // Store num in the dynamic array.
            numCt++;
        }

        TextIO.putln("\nYour numbers in reverse order are:\n");

        for (int i = numCt - 1; i >= 0; i--) {
            TextIO.putln( numbers.get(i) ); // Print the i-th number.
        }

    } // end main();

} // end class ReverseWithDynamicArray

```

The following applet simulates this program. I've included an output statement in the `DynamicArrayOfInt` class. This statement will inform you each time the data array increases in size. (Of course, the output statement doesn't really belong in the class. It's included here for demonstration purposes.)

(Applet "ReverseIntsConsole" would be displayed here
if Java were available.)

ArrayLists

The `DynamicArrayOfInt` class could be used in any situation where an array of `int` with no preset limit on the size is needed. However, if we want to store `Shapes` instead of `ints`, we would have to define a new class to do it. That class, probably named `"DynamicArrayOfShape"`, would look exactly the same as the `DynamicArrayOfInt` class except that everywhere the type `"int"` appears, it would be replaced by the type `"Shape"`. Similarly, we could define a `DynamicArrayOfDouble` class, a `DynamicArrayOfPlayer` class, and so on. But there is something a little silly about this, since all these classes are close to being identical. It would be nice to be able to write some kind of source code, once and for all, that could be used to generate any of these classes on demand, given the type of value that we want to store. This would be an example of **generic programming**. Some programming languages, such as C++, have support for generic programming. Java does not, at least not quite. We can come close to generic programming in Java by working with data structures that contain elements of type `Object`.

In Java, every class is a subclass of the class named `Object`. This means that every object can be assigned to a variable of type `Object`. Any object can be put into an array of type `Object[]`. If a subroutine has a formal parameter of type `Object`, then any object can be passed to the subroutine as an actual parameter. If we defined a `DynamicArrayOfObject` class, then we could store objects of any type. This is not true generic programming, and it doesn't apply to the primitive types such as `int` and `double`. But it does come close. In fact, there is no need for us to define a `DynamicArrayOfObject` class. Java already has a standard class named `ArrayList` that serves much the same purpose. The `ArrayList` class is in the package `java.util`, so if you want to use the `ArrayList` class in a program, you should put the directive `"import java.util.ArrayList;"` or `"import java.util.*;"` at the beginning of your source code file.

The `ArrayList` class differs from my `DynamicArrayOfInt` class in that an `ArrayList` object always has a definite size, and it is illegal to refer to a position in the `ArrayList` that lies outside its size.

In this, an `ArrayList` is more like a regular array. However, the size of an `ArrayList` can be increased at will. The `ArrayList` class defines many instance methods. I'll describe some of the most useful. Suppose that `list` is a variable of type `ArrayList`.

`list.size()` -- This function returns the current size of the `ArrayList`. The only valid positions in the list are numbers in the range 0 to `list.size()-1`. Note that the size can be zero. A call to the default constructor `new ArrayList()` creates an `ArrayList` of size zero.

`list.add(obj)` -- Adds an object onto the end of the `ArrayList`, increasing the size by 1. The parameter, `obj`, can refer to an object of any type, or it can be `null`.

`list.get(N)` -- This function returns the value stored at position `N` in the `ArrayList`. `N` must be an integer in the range 0 to `list.size()-1`. If `N` is outside this range, an error occurs. Calling this function is similar to referring to `A[N]` for an array, `A`, except that you can't use `list.get(N)` on the left side of an assignment statement.

`list.set(N, obj)` -- Assigns the object, `obj`, to position `N` in the `ArrayList`, replacing the item previously stored at position `N`. The integer `N` must be in the range from 0 to `list.size()-1`. A call to this function is equivalent to the command `A[N] = obj` for an array `A`.

`list.remove(obj)` -- If the specified object occurs somewhere in the `ArrayList`, it is removed from the list. Any items in the list that come after the removed item are moved down one position. The size of the `ArrayList` decreases by 1. If `obj` occurs more than once in the list, only the first copy is removed.

`list.remove(N)` -- For an integer, `N`, this removes the `N`-th item in the `ArrayList`. `N` must be in the range 0 to `list.size()-1`. Any items in the list that come after the removed item are moved down one position. The size of the `ArrayList` decreases by 1.

`list.indexOf(obj)` -- A function that searches for the object, `obj`, in the `ArrayList`. If the object is found in the list, then the position number where it is found is returned. If the object is not found, then -1 is returned.

For example, suppose again that players in a game are represented by objects of type `Player`. The players currently in the game could be stored in an `ArrayList` named `players`. This variable would be declared as

```
ArrayList players;
```

and initialized to refer to a new, empty `ArrayList` object with

```
players = new ArrayList();
```

If `newPlayer` is a variable that refers to a `Player` object, the new player would be added to the `ArrayList` and to the game by saying

```
players.add(newPlayer);
```

and if player number `i` leaves the game, it is only necessary to say

```
players.remove(i);
```

Or, if `player` is a variable that refers to the `Player` that is to be removed, you could say

```
players.remove(player);
```

All this works very nicely. The only slight difficulty arises when you use the function `players.get(i)` to get the value stored at position `i` in the `ArrayList`. The return type of this function is `Object`. In this

case the object that is returned by the function is actually of type `Player`. In order to do anything useful with the returned value, it's usually necessary to type-cast it to type `Player`:

```
Player plr = (Player)players.get(i);
```

For example, if the `Player` class includes an instance method `makeMove()` that is called to allow a player to make a move in the game, then the code for letting all the players move is

```
for (int i = 0; i < players.size(); i++) {
    Player plr = (Player)players.get(i);
    plr.makeMove();
}
```

The two lines inside the `for` loop can be combined to a single line:

```
((Player)players.get(i)).makeMove();
```

This gets an item from the list, type-casts it, and then calls the `makeMove()` method on the resulting `Player`. The parentheses around "`(Player)players.get(i)`" are required because of Java's precedence rules. The parentheses force the type-cast to be performed before the `makeMove()` method is called.

In [Section 5.4](#), I displayed an applet, `ShapeDraw`, that uses `ArrayLists`. Here is another version of the same idea, simplified to make it easier to see how `ArrayLists` are being used. Right-click the large white drawing area to add a colored rectangle. (On a Macintosh, Command-click the drawing area.) The color of the rectangle is given by the "rainbow palette" along the bottom of the applet. Click the palette to select a new color. Click and drag rectangles with the left mouse button. Hold down the Alt or Option key and click on a rectangle to delete it. Shift-click a rectangle to move it out in front of all the other rectangles.

(Applet "SimpleDrawRects" would be displayed here
if Java were available.)

The source code for this applet is in the file [SimpleDrawRects.java](#). You should be able to follow it in its entirety. (If you've read [Chapter 7](#), you can also take a look at the file [RainbowPalette.java](#), which defines a custom component that is used for the colored palette in this applet.) Here, I just want to look at the parts of the program that use an `ArrayList`.

The applet uses a variable named `rects`, of type `ArrayList`, to hold information about the rectangles that have been added to the drawing area. The objects that are stored in the list belong to a class, `ColoredRect`, that is defined as

```
class ColoredRect {
    // Holds data for one colored rectangle.
    int x,y;           // Upper left corner of the rectangle.
    int width,height;  // size of the rectangle.
    Color color;       // Color of the rectangle.
}
```

If `g` is a variable of type `Graphics`, then the following code draws all the rectangles that are stored in the list `rects` (with a black outline around each rectangle, as shown in the applet):

```
for (int i = 0; i < rects.size(); i++) {
    ColoredRect rect = (ColoredRect)rects.get(i);
    g.setColor( rect.color );
    g.fillRect( rect.x, rect.y, rect.width, rect.height);
    g.setColor( Color.black );
    g.drawRect( rect.x, rect.y, rect.width - 1, rect.height - 1);
}
```

To implement all the mouse operations in the applet, it must be possible to find the rectangle, if any, that

contains the point where the user clicked the mouse. To do this, I wrote the function

```
ColoredRect findRect(int x, int y) {
    // Find the topmost rect that contains the point (x,y).
    // Return null if no rect contains that point. The
    // rects in the ArrayList are considered in reverse order
    // so that if one lies on top of another, the one on top
    // is seen first and is the one that is returned.

    for (int i = rects.size() - 1; i >= 0; i--) {
        ColoredRect rect = (ColoredRect)rects.get(i);
        if ( x >= rect.x && x < rect.x + rect.width
            && y >= rect.y && y < rect.y + rect.height )
            return rect; // (x,y) is inside this rect.
    }

    return null; // No rect containing (x,y) was found.
}
```

The code for removing a `ColoredRect`, `rect`, from the drawing area is simply `rects.remove(rect)` (followed by a `repaint()`). Bringing a given rectangle out in front of all the other rectangles is just a little harder. Since the rectangles are drawn in the order in which they occur in the `ArrayList`, the rectangle that is in the last position in the list is in front of all the other rectangles on the screen. So we need to move the rectangle to the last position in the list. This is done by removing the rectangle from its current position in the list and then adding it back at the end:

```
void bringToFront(ColoredRect rect) {
    // If rect != null, move it out in front of the other
    // rects by moving it to the last position in the ArrayList.
    if (rect != null) {
        rects.remove(rect);
        rects.add(rect);
        repaint();
    }
}
```

This should be enough to give you the basic idea. You can look in the source code for more details.

Vectors

The `ArrayList` class was introduced in Java version 1.2, as one of a group of classes designed for working with collections of objects. We'll look at these "collection classes" in [Chapter 12](#). Earlier versions of Java did not include `ArrayList`, but they did have a very similar class named `java.util.Vector`. You can still see `Vectors` used in older code and in many of Java's standard classes, so it's worth knowing about them. Using a `Vector` is similar to using an `ArrayList`, except that different names are used for some commonly used instance methods, and some instance methods in one class don't correspond to any instance method in the other class.

Like an `ArrayList`, a `Vector` is similar to an array of `Objects` that can grow to be as large as necessary. The default constructor, `new Vector()`, creates a vector with no elements.

Suppose that `vec` is a `Vector`. Then:

- `vec.size()` is a function that returns the number of elements currently in the vector.
- `vec.addElement(obj)` will add the `Object`, `obj` to the end of the vector. This is the same as the `add()` method of an `ArrayList`.

- `vec.removeElement(obj)` removes `obj` from the vector, if it occurs. Only the first occurrence is removed. This is the same as `remove(obj)` for an `ArrayList`.
- `vec.removeElementAt(N)` removes the `N`-th element, for an integer `N`. `N` must be in the range 0 to `vec.size()-1`. This is the same as `remove(N)` for an `ArrayList`.
- `vec.setSize(N)` sets the size of the vector to `N`. If there were more than `N` elements in `vec`, the extra elements are removed. If there were fewer than `N` elements, extra spaces are filled with `null`. The `ArrayList` class does not have a `setSize()` method.

The `Vector` class includes many more methods, but these are probably the most commonly used.

[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Section 8.4

Searching and Sorting

TWO ARRAY PROCESSING TECHNIQUES that are particularly common are **searching** and **sorting**.

Searching here refers to finding an item in the array that meets some specified criterion. Sorting refers to rearranging all the items in the array into increasing or decreasing order (where the meaning of increasing and decreasing can depend on the context).

Sorting and searching are often discussed, in a theoretical sort of way, using an array of numbers as an example. In practical situations, though, more interesting types of data are usually involved. For example, the array might be a mailing list, and each element of the array might be an object containing a name and address. Given the name of a person, you might want to look up that person's address. This is an example of searching, since you want to find the object in the array that contains the given name. It would also be useful to be able to sort the array according to various criteria. One example of sorting would be ordering the elements of the array so that the names are in alphabetical order. Another example would be to order the elements of the array according to zip code before printing a set of mailing labels. (This kind of sorting can get you a cheaper postage rate on a large mailing.)

This example can be generalized to a more abstract situation in which we have an array that contains objects, and we want to search or sort the array based on the value of one of the instance variables in that array. We can use some terminology here that originated in work with "databases," which are just large, organized collections of data. We refer to each of the objects in the array as a **record**. The instance variables in an object are then called **fields** of the record. In the mailing list example, each record would contain a name and address. The fields of the record might be the first name, last name, street address, state, city and zip code. For the purpose of searching or sorting, one of the fields is designated to be the **key** field. Searching then means finding a record in the array that has a specified value in its key field. Sorting means moving the records around in the array so that the key fields of the record are in increasing (or decreasing) order.

In this section, most of my examples follow the tradition of using arrays of numbers. But I'll also give a few examples using records and keys, to remind you of the more practical applications.

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:

```
static int find(int[] A, int N) {
    // Searches the array A for the integer N.
    // Postcondition: If N is not in the array, -1 is
    //               returned. If N is in the array, then the
    //               return value, i, is the first integer that
    //               satisfies A[i] == N.

    for (int index = 0; index < A.length; index++) {
        if ( A[index] == N )
```

```

        return index;    // N has been found at this index!
    }

    // If we get this far, then N has not been found
    // anywhere in the array.  Return a value of -1.

    return -1;

}

```

This method of searching an array by looking at each item in turn is called **linear search**. If nothing is known about the order of the items in the array, then there is really no better alternative algorithm. But if the elements in the array are known to be in increasing or decreasing order, then a much faster search algorithm can be used. An array in which the elements are in order is said to be **sorted**. Of course, it takes some work to sort an array, but if the array is to be searched many times, then the work done in sorting it can really pay off.

Binary search is a method for searching for a given item in a **sorted array**. Although the implementation is not trivial, the basic idea is simple: If you are searching for an item in a sorted list, then it is possible to eliminate half of the items in the list by inspecting a single item. For example, suppose that you are looking for the number 42 in a sorted array of 1000 integers. Let's assume that the array is sorted into increasing order. Suppose you check item number 500 in the array, and find that the item is 93. Since 42 is less than 93, and since the elements in the array are in increasing order, we can conclude that if 42 occurs in the array at all, then it must occur somewhere before location 500. All the locations numbered 500 or above contain values that are greater than or equal to 93. These locations can be eliminated as possible locations of the number 42.

The next obvious step is to check location 250. If the number at that location is, say, 21, then you can eliminate locations before 250 and limit further search to locations between 251 and 499. The next test will limit the search to about 125 locations, and the one after that to about 62. After just 10 steps, there is only one location left. This is a whole lot better than looking through every element in the array. If there were a million items, it would still take only 20 steps for this method to search the array! (Mathematically, the number of steps is the logarithm, in the base 2, of the number of items in the array.)

In order to make binary search into a Java subroutine that searches an array *A* for an item *N*, we just have to keep track of the range of locations that could possibly contain *N*. At each step, as we eliminate possibilities, we reduce the size of this range. The basic operation is to look at the item in the middle of the range. If this item is greater than *N*, then the second half of the range can be eliminated. If it is less than *N*, then the first half of the range can be eliminated. If the number in the middle just happens to be *N* exactly, then the search is finished. If the size of the range decreases to zero, then the number *N* does not occur in the array. Here is a subroutine that returns the location of *N* in a sorted array *A*. If *N* cannot be found in the array, then a value of -1 is returned instead:

```

static int binarySearch(int[] A, int N) {
    // 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 not, then the
    //      return value is -1.

    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;
}

```

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. The basic operation on association lists is this: Given a key, k , find the value v associated with k , if any.

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. The items in the 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. (This is an example of a "partially full array" as discussed in the [previous section](#). It might be better to use a dynamic array or an `ArrayList` to hold the phone entries.) A phone directory could be an object belonging to the class:

```

class PhoneDirectory {

    PhoneEntry[] info = new PhoneEntry[100]; // Space for 100 entries.
    int entries = 0; // Actual number of entries in the array.

    void addEntry(String name, String phoneNum) {
        // Add a new item at the end of the array.
    }
}

```

```

        info[entries] = new PhoneEntry();
        info[entries].name = name;
        info[entries].phoneNum = phoneNum;
        entries++;
    }

    String getNumber(String name) {
        // Return phone number associated with name,
        // or return null if the name does not occur
        // in the array.
        for (int index = 0; index < entries; index++) {
            if (name.equals( info[index].name )) // Found it!
                return info[index].phoneNum;
        }
        return null; // Name wasn't found.
    }
}

```

Note that the search method, `getNumber`, only looks through the locations in the array that have actually been filled with `PhoneEntries`. Also note that unlike the search routines given earlier, this routine does not return the location of the item in the array. Instead, it returns the value that it finds associated with the key, `name`. This is often done with association lists.

This class could use a lot of improvement. For one thing, it would be nice to use binary search instead of simple linear search in the `getNumber` method. However, we could only do that if the list of `PhoneEntries` were sorted into alphabetical order according to name. In fact, it's really not all that hard to keep the list of entries in sorted order, as you'll see in just a second.

Insertion Sort

We've seen that there are good reasons for sorting arrays. There are many algorithms available for doing so. One of the easiest to understand is the **insertion sort** algorithm. This method is also applicable to the problem of keeping a list in sorted order as you add new items to the list. Let's consider that case first:

Suppose you have a sorted list and you want to add an item to that list. If you want to make sure that the modified list is still sorted, then the item must be inserted into the right location, with all the smaller items coming before it and all the bigger items after it. This will mean moving each of the bigger items up one space to make room for the new item.

```

static void insert(int[] A, int itemsInArray, int newItem) {
    // 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.

    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 list.

        int temp = A[itemsSorted]; // The item to be inserted.
        int loc = itemsSorted - 1; // Start at end of list.

        while (loc >= 0 && A[loc] > temp) {
            A[loc + 1] = A[loc]; // Bump item from A[loc] up to loc+1.
            loc = loc - 1;       // Go on to next location.
        }

        A[loc + 1] = temp; // Put temp in last vacated space.
    }
}

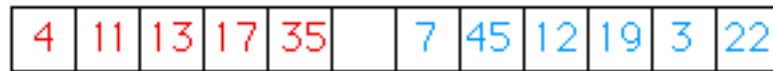
```

The following is an illustration of one stage in insertion sort. It shows what happens during one execution of the `for` loop in the above method, when `itemsSorted` is 5.

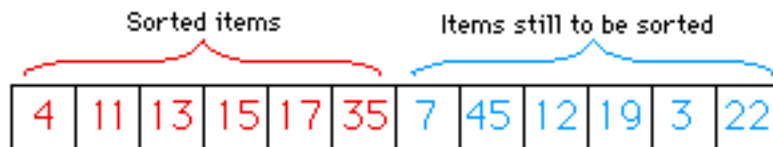
Start with a partially sorted list of items:



Temp: 15 Copy next unsorted item into Temp, leaving a "hole" in the array



Bump any items bigger than Temp up one space, then copy Temp into the "empty" location.



Now, the list of sorted items has increased in size by one item.

Selection Sort

Another typical sorting method uses the idea of finding the biggest item in the list and moving it to the end -- which is where it belongs if the list is to be in increasing order. Once the biggest item is in its correct location, you can then apply the same idea to the remaining items. That is, find the next-biggest item, and move it into the next-to-last space, and so forth. This algorithm is called **selection sort**. It's easy to write:

```
static void selectionSort(int[] A) {

    // Sort A into increasing order, using selection sort

    for (int lastPlace = A.length-1; lastPlace > 0; lastPlace--) {
        // Find the largest item among A[0], A[1], ...,
        // A[lastPlace], and move it into position lastPlace
        // by swapping it with the number that is currently
        // in position lastPlace.

        int maxLoc = 0; // Location of largest item seen so far.

        for (int j = 1; j <= lastPlace; j++) {
            if (A[j] > A[maxLoc]) {
                // Since A[j] is bigger than the maximum we've seen
                // so far, j is the new location of the maximum value
                // we've seen so far.
                maxLoc = j;
            }
        }
    }
}
```



```

    }

    int temp = A[maxLoc]; // Swap largest item with A[lastPlace].
    A[maxLoc] = A[lastPlace];
    A[lastPlace] = temp;

} // end of for loop

}

```

Insertion sort and selection sort are suitable for sorting fairly small arrays (up to a few hundred elements, say). There are more complicated sorting algorithms that are much faster than insertion sort and selection sort for large arrays. I'll discuss one such algorithm in [Section 11.1](#).

A variation of selection sort is used in the `Hand` class that was introduced in [Section 5.3](#). (By the way, you are finally in a position to fully understand the source code for both the `Hand` class and the `Deck` class from that section. See the source files [Deck.java](#) and [Hand.java](#).)

In the `Hand` class, a hand of playing cards is represented by a `Vector`. This is older code, which used `Vector` instead of `ArrayList`, and I have chosen not to modify it so that you would see at least one example of using `Vectors`. See the [previous section](#) for a discussion of `ArrayLists` and `Vectors`.

The objects stored in the `Vector` 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 vector and move the cards one-by-one from the old vector to the new vector. The cards are selected from the old vector in increasing order. In the end, the new vector becomes the hand and the old vector is discarded. This is certainly not an efficient procedure! But hands of cards are so small that the inefficiency is negligible. Here is the code:

```

public void sortBySuit() {
    // 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.
    Vector newHand = new Vector();
    while (hand.size() > 0) {
        int pos = 0; // Position of minimal card.
        Card c = (Card)hand.elementAt(0); // Minimal card seen so far.
        for (int i = 1; i < hand.size(); i++) {
            Card c1 = (Card)hand.elementAt(i);
            if ( c1.getSuit() < c.getSuit() ||
                (c1.getSuit() == c.getSuit()
                 && c1.getValue() < c.getValue()) ) {
                pos = i;
                c = c1;
            }
        }
        hand.removeElementAt(pos);
        newHand.addElement(c);
    }
    hand = newHand;
}

```

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:

```
static void shuffle(int[] A) {  
    // Postcondition:  The items in A have been rearranged into  
    //                  a random order.  
    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;  
    }  
}
```

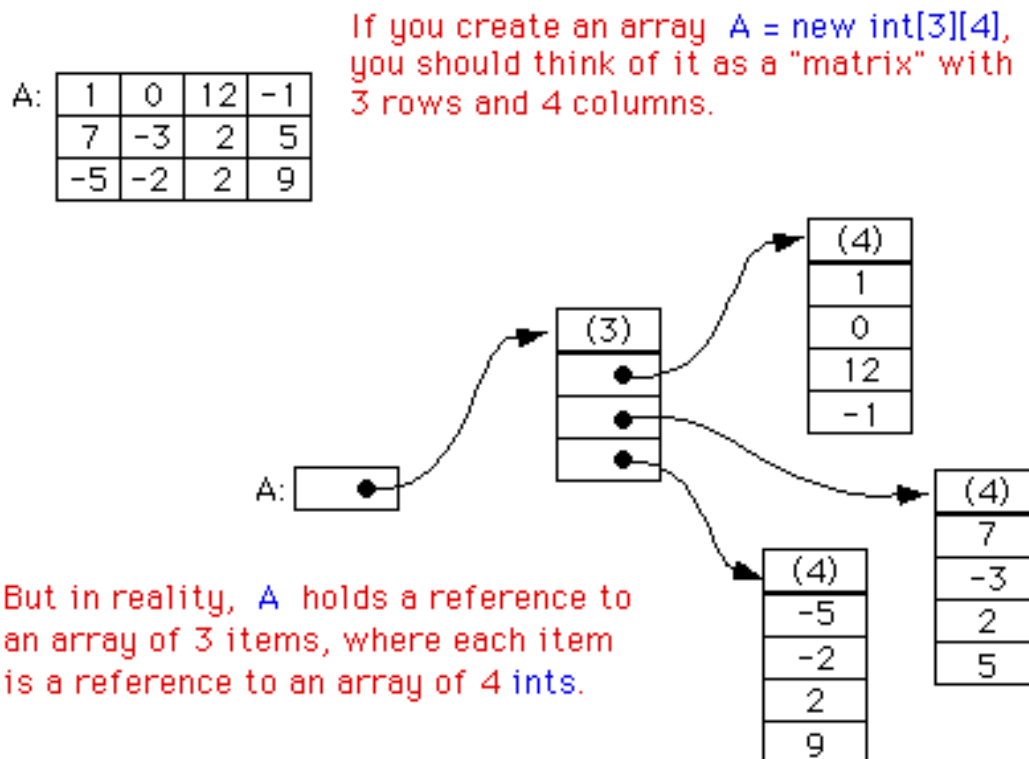
[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Section 8.5

Multi-Dimensional Arrays

ANY TYPE CAN BE USED AS THE BASE TYPE FOR AN ARRAY. You can have an array of `ints`, an array of `Strings`, an array of `Objects`, and so on. In particular, since an array type is a first-class Java type, you can have an array of arrays. For example, an array of `ints` has type `int[]`. This means that there is automatically another type, `int[][]`, which represents an "array of arrays of `ints`". Such an array is said to be a **two-dimensional array**. Of course once you have the type `int[][]`, there is nothing to stop you from forming the type `int[][][]`, which represents a **three-dimensional array** -- and so on. There is no limit on the number of dimensions that an array type can have. However, arrays of dimension three or higher are fairly uncommon, and I concentrate here mainly on two-dimensional arrays. The type `BaseType[][]` is usually read "two-dimensional array of `BaseType`" or "`BaseType` array array".

The declaration statement `int[][] A;` declares a variable named `A` of type `int[][]`. This variable can hold a reference to an object of type `int[][]`. The assignment statement `A = new int[3][4];` creates a new two-dimensional array object and sets `A` to point to the newly created object. As usual, the declaration and assignment could be combined in a single declaration statement `int[][] A = new int[3][4];`. The newly created object is an array of arrays-of-`ints`. The notation `int[3][4]` indicates that there are 3 arrays-of-`ints` in the array `A`, and that there are 4 `ints` in each array-of-`ints`. However, trying to think in such terms can get a bit confusing -- as you might have already noticed. So it is customary to think of a two-dimensional array of items as a rectangular **grid** or **matrix** of items. The notation `"new int[3][4]"` can then be taken to describe a grid of `ints` with 3 rows and 4 columns. The following picture might help:



For the most part, you can ignore the reality and keep the picture of a grid in mind. Sometimes, though, you will need to remember that each row in the grid is really an array in itself. These arrays can be referred to as `A[0]`, `A[1]`, and `A[2]`. Each row is in fact a value of type `int[]`. It could, for example, be passed to a subroutine that asks for a parameter of type `int[]`.

The notation `A[1]` refers to one of the rows of the array `A`. Since `A[1]` is itself an array of `ints`, you can

another subscript to refer to one of the positions in that row. For example, `A[1][3]` refers to item number 3 in row number 1. Keep in mind, of course, that both rows and columns are numbered starting from zero. So, in the above example, `A[1][3]` is 5. More generally, `A[i][j]` refers to the grid position in row number `i` and column number `j`. The 12 items in `A` are named as follows:

<code>A[0][0]</code>	<code>A[0][1]</code>	<code>A[0][2]</code>	<code>A[0][3]</code>
<code>A[1][0]</code>	<code>A[1][1]</code>	<code>A[1][2]</code>	<code>A[1][3]</code>
<code>A[2][0]</code>	<code>A[2][1]</code>	<code>A[2][2]</code>	<code>A[2][3]</code>

`A[i][j]` is actually a variable of type `int`. You can assign integer values to it or use it in any other context where an integer variable is allowed.

It might be worth noting that `A.length` gives the number of rows of `A`. To get the number of columns in `A`, you have to ask how many `ints` there are in a row; this number would be given by `A[0].length`, or equivalently by `A[1].length` or `A[2].length`. (There is actually no rule that says that all the rows of an array must have the same length, and some advanced applications of arrays use varying-sized rows. But if you use the `new` operator to create an array in the manner described above, you'll always get an array with equal-sized rows.)

Three-dimensional arrays are treated similarly. For example, a three-dimensional array of `ints` could be created with the declaration statement `"int[][][] B = new int[7][5][11];"`. It's possible to visualize the value of `B` as a solid 7-by-5-by-11 block of cells. Each cell holds an `int` and represents one position in the three-dimensional array. Individual positions in the array can be referred with variable names of the form `B[i][j][k]`. Higher-dimensional arrays follow the same pattern, although for dimensions greater than three, there is no easy way to visualize the structure of the array.

It's possible to fill a multi-dimensional array with specified items at the time it is declared. Recall that when an ordinary one-dimensional array variable is declared, it can be assigned an "array initializer," which is just a list of values enclosed between braces, `{` and `}`. Array initializers can also be used when a multi-dimensional array is declared. An initializer for a two-dimensional array consists of a list of one-dimensional array initializers, one for each row in the two-dimensional array. For example, the array `A` shown in the picture above could be created with:

```
int[][] A = { { 1, 0, 12, -1 },
               { 7, -3, 2, 5 },
               { -5, -2, 2, 9 }
             };
```

If no initializer is provided for an array, then when the array is created it is automatically filled with the appropriate value: zero for numbers, `false` for boolean, and `null` for objects.

Just as in the case of one-dimensional arrays, two-dimensional arrays are often processed using `for` statements. To process all the items in a two-dimensional array, you have to use one `for` statement nested inside another. If the array `A` is declared as

```
int[][] A = new int[3][4];
```

then you could store a zero into each location in `A` with:

```
for (int row = 0; row < 3; row++) {
    for (int column = 0; column < 4; column++) {
        A[row][column] = 0;
    }
}
```

The first time the outer `for` loop executes (with `row = 0`), the inner `for` loop fills in the four values in the first row of `A`, namely `A[0][0] = 0`, `A[0][1] = 0`, `A[0][2] = 0`, and `A[0][3] = 0`. The next execution of the outer `for` loop fills in the second row of `A`. And the third and final execution of the outer

loop fills in the final row of A.

Similarly, you could add up all the items in A with:

```
int sum = 0;
for (int i = 0; i < 3; i++)
    for (int j = 0; j < 4; i++)
        sum = sum + A[i][j];
```

To process a three-dimensional array, you would, of course, use triply nested `for` loops.

A two-dimensional array can be used whenever the data being represented can be naturally arranged into rows and columns. Often, the grid is built into the problem. For example, a chess board is a grid with 8 rows and 8 columns. If a class named `ChessPiece` is available to represent individual chess pieces, then the contents of a chess board could be represented by a two-dimensional array:

```
ChessPiece[][] board = new ChessPiece[8][8];
```

Or consider the "mosaic" of colored rectangles used as an example in [Section 4.6](#). The mosaic is implemented by a class named `MosaicCanvas`. The data about the color of each of the rectangles in the mosaic is stored in an instance variable named `grid` of type `Color[][]`. Each position in this grid is occupied by a value of type `Color`. There is one position in the grid for each colored rectangle in the mosaic. The actual two-dimensional array is created by the statement:

```
grid = new Color[ROWS][COLUMNS];
```

where `ROWS` is the number of rows of rectangles in the mosaic and `COLUMNS` is the number of columns. The value of the `Color` variable `grid[i][j]` is the color of the rectangle in row number `i` and column number `j`. When the color of that rectangle is changed to some color value, `c`, the value stored in `grid[i][j]` is changed with a statement of the form "`grid[i][j] = c;`". When the mosaic is redrawn, the values stored in the two-dimensional array are used to decide what color to make each rectangle. Here is a simplified version of the code from the `MosaicCanvas` class that draws all the colored rectangles in the grid. You can see how it uses the array:

```
int rowHeight = getSize().height / ROWS;
int colWidth = getSize().width / COLUMNS;
for (int row = 0; row < ROWS; row++) {
    for (int col = 0; col < COLUMNS; col++) {
        g.setColor( grid[row][col] ); // Get color from array.
        g.fillRect( col*colWidth, row*rowHeight,
                    colWidth, rowHeight );
    }
}
```

Sometimes two-dimensional arrays are used in problems in which the grid is not so visually obvious. Consider a company that owns 25 stores. Suppose that the company has data about the profit earned at each store for each month in the year 2000. If the stores are numbered from 0 to 24, and if the twelve months from January '00 through December '00 are numbered from 0 to 11, then the profit data could be stored in an array, `profit`, constructed as follows:

```
double[][] profit = new double[25][12];
```

`profit[3][2]` would be the amount of profit earned at store number 3 in March, and more generally, `profit[storeNum][monthNum]` would be the amount of profit earned in store number `storeNum` in month number `monthNum`. In this example, the one-dimensional array `profit[storeNum]` has a very useful meaning: It is just the profit data for one particular store for the whole year.

Let's assume that the `profit` array has already been filled with data. This data can be processed in a lot of interesting ways. For example, the total profit for the company -- for the whole year from all its stores -- can

be calculated by adding up all the entries in the array:

```
double totalProfit; // Company's total profit in 2000.

totalProfit = 0;
for (int store = 0; store < 25; store++) {
    for (int month = 0; month < 12; month++)
        totalProfit += profit[store][month];
}
```

Sometimes it is necessary to process a single row or a single column of an array, not the entire array. For example, to compute the total profit earned by the company in December, that is, in month number 11, you could use the loop:

```
double decemberProfit = 0.0;
for (storeNum = 0; storeNum < 25; storeNum++)
    decemberProfit += profit[storeNum][11];
```

Let's extend this idea to create a one-dimensional array that contains the total profit for each month of the year:

```
double[] monthlyProfit; // Holds profit for each month.
monthlyProfit = new double[12];

for (int month = 0; month < 12; month++) {
    // compute the total profit from all stores in this month.
    monthlyProfit[month] = 0.0;
    for (int store = 0; store < 25; store++) {
        // Add the profit from this store in this month
        // into the total profit figure for the month.
        monthlyProfit[month] += profit[store][month];
    }
}
```

As a final example of processing the profit array, suppose that we wanted to know which store generated the most profit over the course of the year. To do this, we have to add up the monthly profits for each store. In array terms, this means that we want to find the sum of each row in the array. As we do this, we need to keep track of which row produces the largest total.

```
double maxProfit; // Maximum profit earned by a store.
int bestStore;    // The number of the store with the
                  // maximum profit.

double total = 0.0; // Total profit for one store.

// First compute the profit from store number 0.

for (int month = 0; month < 12; month++)
    total += profit[0][month];

bestStore = 0; // Start by assuming that the best
maxProfit = total; // store is store number 0.

// Now, go through the other stores, and whenever we
// find one with a bigger profit than maxProfit, revise
// the assumptions about bestStore and maxProfit.

for (store = 1; store < 25; store++) {
```

```

        // Compute this store's profit for the year.

        total = 0.0;
        for (month = 0; month < 12; month++)
            total += profit[store][month];

        // Compare this store's profits with the highest
        // profit we have seen among the preceding stores.

        if (total > maxProfit) {
            maxProfit = total;    // Best profit seen so far!
            bestStore = store;    // It came from this store.
        }

    } // end for

    // At this point, maxProfit is the best profit of any
    // of the 25 stores, and bestStore is a store that
    // generated that profit. (Note that there could also be
    // other stores that generated exactly the same profit.)

```

For the rest of this section, we'll look at a more substantial example. Here is an applet that lets two users play checkers against each other. A player moves by clicking on the piece to be moved and then on the empty square to which it is to be moved. The squares that the current player can legally click are hilited. A piece that has been selected to be moved is surrounded by a white border. Other pieces that can legally be moved are surrounded by a cyan-colored border. If a piece has been selected, each empty square that it can legally move to is hilited 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.

(Applet "Checkers" would be displayed here
if Java were available.)

I will only cover a part of the programming of this applet. I encourage you to read the complete source code, [Checkers.java](#). At over 700 lines, this is a more substantial example than anything you've seen before in this course, but it's an excellent example of state-based, event-driven programming. The source file defines four classes. The logic of the game is implemented in a class named CheckersCanvas.

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. One of these classes is CheckersData, which handles the data for the board. It is mainly this class that I want to talk about.

The CheckersData class has an instance variable named board of type `int[][]`. The value of board is set to "new `int[8][8]`", an 8-by-8 grid of integers. The values stored in the grid are defined as constants representing the possible contents of a square on a checkerboard:

```

public static final int
    EMPTY = 0,           // Value representing an empty square.
    RED = 1,             // A regular red piece.
    RED_KING = 2,        // A red king.
    BLACK = 3,           // A regular black piece.
    BLACK_KING = 4;      // A black king.

```

The constants RED and BLACK are also used in my program (or, perhaps, misused) to represent the two players in the game. When a game is started, the values in the variable, board, are set to represent the initial state of the board. The grid of values looks like

	0	1	2	3	4	5	6	6
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 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 red piece can only move up the grid. Kings of either color, of course, can move in both directions.

One function of the `CheckersData` class is to take care of all the details of making moves on the board. An instance method named `makeMove()` is provided to do this. When a player moves a piece from one square to another, the values stored at two positions in the array are changed. But that's not all. If the move is a jump, then the piece that was jumped is removed from the board. (The method checks whether the move is a jump by checking if the square to which the piece is moving is two rows away from the square where it starts.) Furthermore, a RED piece that moves to row 0 or a BLACK piece that moves to row 7 becomes a king. This is good programming: the rest of the program doesn't have to worry about any of these details. It just calls this `makeMove()` method:

```
public void makeMove(int fromRow, int fromCol, int toRow, int 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.

    board[toRow][toCol] = board[fromRow][fromCol]; // Move the piece.
    board[fromRow][fromCol] = EMPTY;

    if (fromRow - toRow == 2 || fromRow - toRow == -2) {
        // The move is a jump. Remove the jumped piece from the board.
        int jumpRow = (fromRow + toRow) / 2; // Row of the jumped piece.
        int jumpCol = (fromCol + toCol) / 2; // Column of the jumped piece.
        board[jumpRow][jumpCol] = EMPTY;
    }

    if (toRow == 0 && board[toRow][toCol] == RED)
        board[toRow][toCol] = RED_KING; // Red piece becomes a king.
    if (toRow == 7 && board[toRow][toCol] == BLACK)
        board[toRow][toCol] = BLACK_KING; // Black piece becomes a king.
} // end makeMove()
```

An even more important function of the CheckersData class is to find legal moves on the board. In my program, a move in a Checkers game is represented by an object belonging to the following class:

```
class CheckersMove {
    // 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.)

    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

```
public CheckersMove[] getLegalMoves(int player)
    // 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.
```

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 a `ArrayList` instead of an array to hold the moves as we find them. 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
Find any legal jumps and add them to moves
if moves.size() is 0:
    Find any other legal moves and add them to moves
if moves.size() is 0:
    return null
else:
    Let moveArray be an array of CheckersMoves of length moves.size()
    Copy the contents of moves into moveArray
    return moveArray

```

Now, how do we find the legal jumps or the legal moves? The information we need is in the board array, but it takes some work to extract it. We have to look through all the positions in the array and find the pieces that belong to the current player. For each piece, we have to check each square that it could conceivably move to, and check whether that would be a legal move. There are four squares to consider. For a jump, we want to look at squares that are two rows and two columns away from the piece. 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:

```

private boolean canMove(int player, int r1, int c1, int r2, int c2) {
    // 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.

    if (r2 < 0 || r2 >= 8 || c2 < 0 || c2 >= 8)
        return false; // (r2,c2) is off the board.

    if (board[r2][c2] != EMPTY)
        return false; // (r2,c2) already contains a piece.

    if (player == RED) {
        if (board[r1][c1] == RED && r2 > r1)
            return false; // Regular red piece can only move down.
    }
}

```

```

        return true; // The move is legal.
    }
    else {
        if (board[r1][c1] == BLACK && r2 < r1)
            return false; // Regular black piece can only move up.
        return true; // The move is legal.
    }
} // end canMove()

```

This method is called by my `getLegalMoves()` method to check whether one of the possible moves that it has found is actually legal. I have a similar method that is called to check whether a jump is legal. In this case, I pass to the method the square containing the player's piece, the square that the player might move to, and the square between those two, which the player would be jumping over. The square that is being jumped must contain one of the opponent's pieces. This method has the specification:

```

private boolean canJump(int player, int r1, int c1,
                       int r2, int c2, int r3, int c3) {
    // 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).
}

```

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:

```

public CheckersMove[] getLegalMoves(int player) {

    if (player != RED && player != BLACK)
        return null;

    int playerKing; // The constant for a King belonging to the player.
    if (player == RED)
        playerKing = RED_KING;
    else
        playerKing = BLACK_KING;

    ArrayList moves = new ArrayList();
        // 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] = (CheckersMove)moves.get(i);
    return moveArray;
}

} // end getLegalMoves

```

End of Chapter 8

[[Next Chapter](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Programming Exercises For Chapter 8

THIS PAGE CONTAINS programming exercises based on material from [Chapter 8](#) of this [on-line Java textbook](#). Each exercise has a link to a discussion of one possible solution of that exercise.

Exercise 8.1: An example in [Section 8.2](#) tried to answer the question, How many random people do you have to select before you find a duplicate birthday? The source code for that program can be found in the file [BirthdayProblemDemo.java](#). Here are some related questions:

- How many random people do you have to select before you find three people who share the same birthday? (That is, all three people were born on the same day in the same month, but not necessarily in the same year.)
- Suppose you choose 365 people at random. How many different birthdays will they have? (The number could theoretically be anywhere from 1 to 365).
- How many different people do you have to check before you've found at least one person with a birthday on each of the 365 days of the year?

Write three programs to answer these questions. Like the example program, `BirthdayProblemDemo`, 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 8.2: Write a program that will read a sequence of positive real numbers entered by the user and will print the same numbers in sorted order from smallest to largest. The user will input a zero to mark the end of the input. Assume that at most 100 positive numbers will be entered.

[See the solution!](#)

Exercise 8.3: A **polygon** is a geometric figure made up of a sequence of connected line segments. The points where the line segments meet are called the **vertices** of the polygon. The `Graphics` class includes commands for drawing and filling polygons. For these commands, the coordinates of the vertices of the polygon are stored in arrays. If `g` is a variable of type `Graphics` then

- `g.drawPolygon(xCoords, yCoords, pointCt)` will draw the outline of the polygon with vertices at `(xCoords[0], yCoords[0])`, `(xCoords[1], yCoords[1])`, ..., `(xCoords[pointCt-1], yCoords[pointCt-1])`. The third parameter, `pointCt`, is an `int` that specifies the number of vertices of the polygon. Its value should be 3 or greater. The first two parameters are arrays of type `int[]`. Note that the polygon automatically includes a line from the last point, `(xCoords[pointCt-1], yCoords[pointCt-1])`, back to the starting point `(xCoords[0], yCoords[0])`.
- `g.fillPolygon(xCoords, yCoords, pointCt)` fills the interior of the polygon with the current drawing color. The parameters have the same meaning as in the `drawPolygon()` method. Note that it is OK for the sides of the polygon to cross each other, but the interior of a polygon with self-intersections might not be exactly what you expect.

Write a little applet that lets the user draw polygons. As the user clicks a sequence of points, count them and store their x- and y-coordinates in two arrays. These points will be the vertices of the polygon. Also, draw a

line between each consecutive pair of points to give the user some visual feedback. When the user clicks near the starting point, draw the complete polygon. Draw it with a red interior and a black border. The user should then be able to start drawing a new polygon. When the user shift-clicks on the applet, clear it.

There is no need to store information about the contents of the applet. The `paintComponent()` method can just draw a border around the applet. The lines and polygons can be drawn using a graphics context, `g`, obtained with the command `"g = getGraphics();"`.

You can try my solution. Note that as the user is drawing the polygon, lines are drawn between the points that the user clicks. Click within two pixels of the starting point to see a filled polygon.

[See the solution!](#)

Exercise 8.4: For this problem, you will need to use an array of objects. The objects belong to the class `MovingBall`, which I have already written. You can find the source code for this class in the file [MovingBall.java](#). A `MovingBall` represents a circle that has an associated color, radius, direction, and speed. It is restricted to moving in a rectangle in the (x, y) plane. It will "bounce back" when it hits one of the sides of this rectangle. A `MovingBall` does not actually move by itself. It's just a collection of data. You have to call instance methods to tell it to update its position and to draw itself. The constructor for the `MovingBall` class takes the form

```
new MovingBall(xmin, xmax, ymin, ymax)
```

where the parameters are integers that specify the limits on the x and y coordinates of the ball. In this exercise, you will want balls to bounce off the sides of the applet, so you will create them with the constructor call `"new MovingBall(0, getWidth(), 0, getHeight())"`. The constructor creates a ball that initially is colored red, has a radius of 5 pixels, is located at the center of its range, has a random speed between 4 and 12, and is headed in a random direction. If `ball` is a variable of type `MovingBall`, then the following methods are available:

- `ball.draw(g)` -- draw the ball in a graphics context. The parameter, `g`, must be of type `Graphics`. (The drawing color in `g` will be changed to the color of the ball.)
- `ball.travel()` -- change the (x, y) -coordinates of the ball by an amount equal to its speed. The ball has a certain direction of motion, and the ball is moved in that direction. Ordinarily, you will call this once for each frame of an animation, so the speed is given in terms of "pixels per frame". Calling this routine does not move the ball on the screen. It just changes the values of some instance variables in the object. The next time the object's `draw()` method is called, the ball will be drawn in the new position.
- `ball.headTowards(x, y)` -- change the direction of motion of the ball so that it is headed towards the point (x, y) . This does not affect the speed.

These are the methods that you will need for this exercise. There are also methods for setting various properties of the ball, such as `ball.setColor(color)` for changing the color and `ball.setRadius(radius)` for changing its size. See the source code for more information.

For this exercise, you should create an applet that shows an animation of 25 balls bouncing around on a black background. Your applet can be defined as a subclass of [SimpleAnimationApplet2](#), which was first introduced in [Section 3.7](#). The `drawFrame()` method in your applet should move all the balls and draw them. (Alternatively, if you have read Chapter 7, you can program the animation yourself using a `Timer`.) Use an array of type `MovingBall[]` to hold the 25 balls.

In addition, your applet should implement the `MouseListener` and `MouseMotionListener` interfaces. When the user presses the mouse or drags the mouse, call each of the ball's `headTowards()` methods to make the balls head towards the mouse's location.

Here is my solution. Try clicking and dragging on the applet:

[See the solution!](#)

Exercise 8.5: The game of Go Moku (also known as Pente or Five Stones) is similar to Tic-Tac-Toe, except that it played on a much larger board and the object is to get five squares in a row rather than three. Players take turns placing pieces on a board. A piece can be placed in any empty square. The first player to get five pieces in a row -- horizontally, vertically, or diagonally -- wins. If all squares are filled before either player wins, then the game is a draw. Write an applet that lets two players play Go Moku against each other.

Your applet will be simpler than the `Checkers` applet from [Section 8.5](#). Play alternates strictly between the two players, and there is no need to highlight the legal moves. You will only need two classes, a short applet class to set up the applet and a `Board` class to draw the board and do all the work of the game. Nevertheless, you will probably want to look at the source code for the checkers applet, [Checkers.java](#), for ideas about the general outline of the program.

The hardest part of the program is checking whether the move that a player makes is a winning move. To do this, you have to look in each of the four possible directions from the square where the user has placed a piece. You have to count how many pieces that player has in a row in that direction. If the number is five or more in any direction, then that player wins. As a hint, here is part of the code from my applet. This code counts the number of pieces that the user has in a row in a specified direction. The direction is specified by two integers, `dirX` and `dirY`. The values of these variables are 0, 1, or -1, and at least one of them is non-zero. For example, to look in the horizontal direction, `dirX` is 1 and `dirY` is 0.

```
int ct = 1; // Number of pieces in a row belonging to the player.

int r, c; // A row and column to be examined.

r = row + dirX; // Look at square in specified direction.
c = col + dirY;
while ( r >= 0 && r < 13 && c >= 0 && c < 13
        && board[r][c] == player ) {
    // Square is on the board, and it
    // contains one of the players'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 my applet. It uses a 13-by-13 board. You can do the same or use a normal 8-by-8 checkerboard.

[See the solution!](#)

Quiz Questions For Chapter 8

THIS PAGE CONTAINS A SAMPLE quiz on material from [Chapter 8](#) of this [on-line Java textbook](#). You should be able to answer these questions after studying that chapter. Sample answers to all the quiz questions can be found [here](#).

Question 1: What does the computer do when it executes the following statement? Try to give as complete an answer as possible.

```
Color[] palette = new Color[12];
```

Question 2: What is meant by the *basetype* of an array?

Question 3: What does it mean to sort an array?

Question 4: What is meant by a *dynamic array*? What is the advantage of a dynamic array over a regular array?

Question 5: What is the purpose of the following subroutine? What is the meaning of the value that it returns, in terms of the value of its parameter?

```
static String concat( String[] str ) {
    if (str == null)
        return null;
    String ans = "";
    for (int i = 0; i < str.length; i++) {
        ans = ans + str[i];
    }
    return ans;
}
```

Question 6: 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 7: Write a complete subroutine that finds the largest value in an array of `ints`. The subroutine 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 subroutine.

Question 8: Suppose that temperature measurements were made on each day of 1999 in each of 100 cities. The measurements have been stored in an array

```
int[][] temps = new int[100][365];
```

where `temps[c][d]` holds the measurement for city number `c` on the `d`th day of the year. Write a code segment that will print out the average temperature, over the course of the whole year, for each city. The average temperature for a city can be obtained by adding up all 365 measurements for that city and dividing the answer by 365.0.

Question 9: 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.

Question 10: Suppose that `A` has been declared and initialized with the statement

```
double[] A = new double[20];
```

And suppose that `A` has already been filled with 20 values. Write a program segment that will find the average of all the non-zero numbers in the array. (The average is the sum of the numbers, divided by the number of numbers. Note that you will have to count the number of non-zero entries in the array.) Declare any variables that you use.

[[Answers](#) | [Chapter Index](#) | [Main Index](#)]

Chapter 9

Correctness and Robustness

COMPUTER PROGRAMS THAT FAIL are much too common. Programs are fragile. A tiny error can cause a program to misbehave or crash. Most of us are familiar with this from our own experience with computers. And we've all heard stories about software glitches that cause spacecraft to crash, telephone service to fail, and, in a few cases, people to die.

Programs don't have to be as bad as they are. It might well be impossible to guarantee that programs are problem-free, but careful programming and well-designed programming tools can help keep the problems to a minimum. This chapter will look at issues of correctness and robustness of programs. We'll also look at **exceptions**, one of the tools that Java provides as an aid in writing robust programs.

Contents of Chapter 9:

- Section 1: [Introduction to Correctness and Robustness](#)
- Section 2: [Writing Correct Programs](#)
- Section 3: [Exceptions and the `try...catch` Statement](#)
- Section 4: [Programming with Exceptions](#)
- [Programming Exercises](#)
- [Quiz on this Chapter](#)

[[First Section](#) | [Next Chapter](#) | [Previous Chapter](#) | [Main Index](#)]

Section 9.1

Introduction to Correctness and Robustness

A PROGRAM IS **correct** if 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 at all 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.

Most computer users have personal experience with programs that don't work or that crash. In many cases, such problems are just annoyances, but even on a personal computer there can be more serious consequences, such as lost work or lost money. When computers are given more important tasks, the consequences of failure can be proportionately more serious.

Just a few years ago, the failure of two multi-million space missions to Mars was prominent in the news. Both failures were probably due to software problems, but in both cases the problem was not with an incorrect program as such. In September 1999, the Mars Climate Orbiter burned up in the Martian atmosphere because data that was expressed in English units of measurement (such as feet and pounds) was entered into a computer program that was designed to use metric units (such as centimeters and grams). A few months later, the Mars Polar Lander probably crashed because its software turned off its landing engines too soon. The program was supposed to detect the bump when the spacecraft landed and turn off the engines then. It has been determined that deployment of the landing gear might have jarred the spacecraft enough to activate the program, causing it to turn off the engines when the spacecraft was still in the air. The unpowered spacecraft would then have fallen to the Martian surface. A more robust system would have checked the altitude before turning off the engines!

There are many equally dramatic stories of problems caused by incorrect or poorly written software. Let's look at a few incidents recounted in the book *Computer Ethics* by Tom Forester and Perry Morrison. (This book covers various ethical issues in computing. It, or something like it, is essential reading for any student of computer science.)

In 1985 and 1986, one person was killed and several were injured by excess radiation, while undergoing radiation treatments by a mis-programmed computerized radiation machine. In another case, over a ten-year period ending in 1992, almost 1,000 cancer patients received radiation dosages that were 30% less than prescribed because of a programming error.

In 1985, a computer at the Bank of New York started destroying records of on-going security transactions because of an error in a program. It took less than 24 hours to fix the program, but by that time, the bank was out \$5,000,000 in overnight interest payments on funds that it had to borrow to cover the problem.

The programming of the inertial guidance system of the F-16 fighter plane would have turned the plane upside-down when it crossed the equator, if the problem had not been discovered in simulation. The Mariner 18 space probe was lost because of an error in one line of a program. The Gemini V space capsule missed its scheduled landing target by a hundred miles, because a programmer forgot to take into account the rotation of the Earth.

In 1990, AT&T's long-distance telephone service was disrupted throughout the United States when a newly loaded computer program proved to contain a bug.

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.

Part of the problem, according to the inventors of Java, can be traced to programming languages themselves. Java was designed to provide some protection against certain types of errors. How can a language feature help prevent errors? Let's look at a few examples.

Early programming languages did not require variables to be declared. In such languages, when a variable name is used in a program, the variable is created automatically. You might consider this more convenient than having to declare every variable explicitly. But there is an unfortunate consequence: An inadvertent spelling error might introduce an extra variable that you had no intention of creating. This type of error was responsible, according to one famous story, for yet another lost spacecraft. In the FORTRAN programming language, the command "DO 20 I = 1, 5" is the first statement of a 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](#)). In C or C++, the computer will just go ahead and store the data in memory that is not part of the array. Since there is no telling what that memory location is being used for, the result will be unpredictable. The consequences could be much more serious than a terminated program. (See, for example, the discussion of buffer overflow errors later in this section.)

Pointers are a notorious source of programming errors. In Java, a variable of object type holds either a pointer to an object or the special value `null`. Any attempt to use a `null` value as if it were a pointer to an actual object will be detected by the system. In some other languages, again, it's up to the programmer to avoid such null pointer errors. In my old Macintosh computer, a `null` pointer was actually implemented as if it were a pointer to memory location zero. A program could use a null pointer to change values stored in

memory near location zero. Unfortunately, the Macintosh stored important system data in those locations. Changing that data could cause the whole system to crash, a consequence more severe than a single failed program.

Another type of pointer error occurs when a pointer value is pointing to an object of the wrong type or to a segment of memory that does not even hold a valid object at all. These types of errors are impossible in Java, which does not allow programmers to manipulate pointers directly. In other languages, it is possible to set a pointer to point, essentially, to any location in memory. If this is done incorrectly, then using the pointer can have unpredictable results.

Another type of error that cannot occur in Java is a memory leak. In Java, once there are no longer any pointers that refer to an object, that object is "garbage collected" so that the memory that it occupied can be reused. In other languages, it is the programmer's responsibility to return unused memory to the system. If the programmer fails to do this, unused memory can build up, leaving less memory for programs and data. There is a story that many common programs for Windows computers have so many memory leaks that the computer will run out of memory after a few days of use and will have to be restarted.

Many programs have been found to suffer from **buffer overflow errors**. Buffer overflow errors often make the news because they are responsible for many network security problems. When one computer receives data from another computer over a network, that data is stored in a buffer. The buffer is just a segment of memory that has been allocated by a program to hold data that it expects to receive. A buffer overflow occurs when more data is received than will fit in the buffer. The question is, what happens then? If the error is detected by the program or by the networking software, then the only thing that has happened is a failed network data transmission. The real problem occurs when the software does not properly detect buffer overflows. In that case, the software continues to store data in memory even after the buffer is filled, and the extra data goes into some part of memory that was not allocated by the program as part of the buffer. That memory might be in use for some other purpose. It might contain important data. It might even contain part of the program itself. This is where the real security issues come in. Suppose that a buffer overflow causes part of a program to be replaced with extra data received over a network. When the computer goes to execute the part of the program that was replaced, it's actually executing data that was received from another computer. That data could be anything. It could be a program that crashes the computer or takes it over. A malicious programmer who finds a convenient buffer overflow error in networking software can try to exploit that error to trick other computers into executing his programs.

For software written completely in Java, buffer overflow errors are impossible. The language simply does not provide any way to store data into memory that has not been properly allocated. To do that, you would need a pointer that points to unallocated memory or you would have to refer to an array location that lies outside the range allocated for the array. As explained above, neither of these is possible in Java. (However, there could conceivably still be errors in Java's standard classes, since some of the methods in these classes are actually written in the C programming language rather than in Java.)

It's clear that language design can help prevent errors or detect them when they occur. Doing so involves restricting what a programmer is allowed to do. Or it requires tests, such as checking whether a pointer is `null`, that take some extra processing time. Some programmers feel that the sacrifice of power and efficiency is too high a price to pay for the extra security. In some applications, this is true. However, there are many situations where safety and security are primary considerations. Java is designed for such situations.

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 first introduced in [Section 3.2](#). Starting from a positive integer N , the program computes a certain sequence of integers:

```
while ( N != 1 ) {
    if ( N % 2 == 0 )    // If N is even...
        N = N / 2;
    else
        N = 3 * N + 1;
    System.out.println(N);
}
```

But there is a problem here: If N is too large, then the value of $3*N+1$ will not be mathematically correct because of integer overflow. The problem arises whenever $3*N+1 > 2147483647$, that is when $N > 2147483646/3$. For a completely correct program, we should check for this possibility **before** computing $3*N+1$:

```
while ( N != 1 ) {
    if ( N % 2 == 0 )    // If N is even...
        N = N / 2;
    else {
        if ( N > 2147483646/3 ) {
            System.out.println("Sorry, but the value of N has become");
            System.out.println("too large for your computer!");
            break;
        }
        N = 3 * N + 1;
    }
    System.out.println(N);
}
```

The problem here is not that the original algorithm for computing $3N+1$ sequences was wrong. The problem is that it just can't be correctly implemented using 32-bit integers. Many programs ignore this type of problem. But integer overflow errors have been responsible for their share of serious computer failures, and a completely robust program should take the possibility of integer overflow into account. (The infamous "Y2K" bug was, in fact, just this sort of error.)

For numbers of type `double`, there are even more problems. There are still overflow errors, which occur when the result of a computation is outside the range of values that can be represented as a value of type `double`. This range extends up to about 1.7 times 10 to the power 308 . Numbers beyond this range do not "wrap around" to negative values. Instead, they are represented by special values that have no numerical equivalent. The 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 by zero or taking the square root of a negative number is `Double.NaN`. You can test whether a number x is this special non-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.

Not all possible errors are 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 programmer still needs to learn techniques for avoiding and dealing with errors. These are the topics of the rest of this chapter.

[[Next Section](#) | [Previous Chapter](#) | [Chapter Index](#) | [Main Index](#)]

Section 9.2

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.

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 the execution of the program. The state includes, for example, the values of all the variables in the program, the output that has been produced, any input that is waiting to be read, and a record of the position in the program where the computer is working. A process is the sequence of states that the computer goes through as it executes the program. From this point of view, the meaning of a statement in a program can be expressed in terms of the effect that the execution of that statement has on the computer's state. As a simple example, the meaning of the assignment statement "`x = 7;`" is that after this statement is executed, the value of the variable `x` will be 7. We can be absolutely sure of this fact, so it is something upon which we can build part of a mathematical proof.

In fact, it is often possible to look at a program and deduce that some fact must be true at a given point during the execution of a program. For example, consider the `do` loop

```
do {
    TextIO.put("Enter a positive integer: ");
    N = TextIO.getlnInt();
} while (N <= 0);
```

After this loop ends, we can be absolutely sure that the value of the variable `N` is greater than zero. The loop cannot end until this condition is satisfied. This fact is part of the meaning of the `while` loop. More generally, if a `while` loop uses the test "`while (condition)`", then after the loop ends, we can be sure that the **condition** is false. We can then use this fact to draw further deductions about what happens as the execution of the program continues. (With a loop, by the way, we also have to worry about the question of whether the loop will ever end. This is something that has to be verified separately.)

A fact that can be proven to be true after a given program segment has been executed is called a **postcondition** of that program segment. Postconditions are known facts upon which we can build further deductions about the behavior of the program. A postcondition of a program as a whole is simply a fact that can be proven to be true after the program has finished executing. A program can be proven to be correct by showing that the postconditions of the program meet the program's specification.

Consider the following program segment, where all the variables are of type `double`:

```
disc = B*B - 4*A*C;
x = (-B + Math.sqrt(disc)) / (2*A);
```

The quadratic formula (from high-school mathematics) assures us that the value assigned to `x` is a solution of the equation $Ax^2 + Bx + C = 0$, provided that the value of `disc` is greater than or equal to zero and the value of `A` is not zero. If we can assume or guarantee that $B^2 - 4AC \geq 0$ and that $A \neq 0$, then

the fact that x is a solution of the equation becomes a postcondition of the program segment. We say that the condition, $B*B-4*A*C \geq 0$ is a **precondition** of the program segment. The condition that $A \neq 0$ is another precondition. A precondition is defined to be 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 [Section 4.6](#). That section introduced preconditions and postconditions as a way of specifying the contract of a subroutine. As the terms are being used here, a precondition of a subroutine is just a precondition of the code that makes up the definition of the subroutine, and the postcondition of a subroutine is a postcondition of the same code. In this section, we have generalized these terms to make them more useful in talking about program correctness.

Let's see how this works by considering a longer program segment:

```
do {
    TextIO.putln("Enter A, B, and C.  B*B-4*A*C must be >= 0.");
    TextIO.put("A = ");
    A = TextIO.getlnDouble();
    TextIO.put("B = ");
    B = TextIO.getlnDouble();
    TextIO.put("C = ");
    C = TextIO.getlnDouble();
    if (A == 0 || B*B - 4*A*C < 0)
        TextIO.putln("Your input is illegal.  Try again.");
} while (A == 0 || B*B - 4*A*C < 0);

disc = B*B - 4*A*C;
x = (-B + Math.sqrt(disc)) / (2*A);
```

After the loop ends, we can be sure that $B*B-4*A*C \geq 0$ and that $A \neq 0$. 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 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 at the end of the [previous section](#).)

Here is another variation, in which the precondition is checked by an `if` statement. In the first part of the `if` statement, where a solution is computed and printed, we know that the preconditions are fulfilled. In the other parts, we know that one of the preconditions fails to hold. In any case, the program is correct.

```
TextIO.putln("Enter your values for A, B, and C.");
TextIO.put("A = ");
A = TextIO.getlnDouble();
TextIO.put("B = ");
B = TextIO.getlnDouble();
TextIO.put("C = ");
C = TextIO.getlnDouble();

if (A != 0 && B*B - 4*A*C >= 0) {
    disc = B*B - 4*A*C;
    x = (-B + Math.sqrt(disc)) / (2*A);
    TextIO.putln("A solution of A*X*X + B*X + C = 0 is " + x);
}
else if (A == 0) {
    TextIO.putln("The value of A cannot be zero.");
}
```

```

    else {
        TextIO.putln("Since B*B - 4*A*C is less than zero, the");
        TextIO.putln("equation A*X*X + B*X + C = 0 has no solution.");
    }

```

Whenever you write a program, it's a good idea to watch out for preconditions and think about how your program handles them. Often, a precondition can offer a clue about how to write the program.

For example, every array reference, such as `A[i]`, has a precondition: The index must be within the range of legal indices for the array. For `A[i]`, the precondition is that $0 \leq i < A.length$. 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`:

```

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);

```

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 the [next chapter](#), which will make essential use of material that will be covered in the next two sections 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's input is not a legal value, then `TextIO` will ask the user to re-enter it. 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.)

```
static void skipBlanks() {
    // 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.
    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()
```

An example in [Section 3.5](#) allowed the user to enter length measurements such as "3 miles" or "1 foot". It would then convert the measurement into inches, feet, yards, and miles. But people commonly use combined measurements such as "3 feet 7 inches". Let's improve the program so that it allows inputs of this form.

More specifically, the user will input lines containing one or more measurements such as "1 foot" or "3 miles 20 yards 2 feet". The legal units of measure are inch, foot, yard, and mile. The program will also recognize plurals (inches, feet, yards, miles) and abbreviations (in, ft, yd, mi). Let's write a subroutine that will read one line of input of this form and compute the equivalent number of inches. The main program uses the number of inches to compute the equivalent number of feet, yards, and miles. If there is any error in the input, the subroutine will print an error message and return the value -1. The subroutine assumes that the input line is not empty. The main program tests for this before calling the subroutine and uses an empty line as a signal for ending the program.

Ignoring the possibility of illegal inputs, a pseudocode algorithm for the subroutine is

```
inches = 0    // This will be the total number of inches
while there is more input on the line:
    read the numerical measurement
    read the units of measure
    add the measurement to inches
return inches
```

We can test whether there is more input on the line by checking whether the next non-blank character is the end-of-line character. But this test has a precondition: Before we can test the next non-blank character, we have to skip over any blanks. So, the algorithm becomes

```
inches = 0
skipBlanks()
while TextIO.peek() is not '\n':
    read the numerical measurement
```



```

        read the unit of measure
        add the measurement to inches
        skipBlanks()
    return inches

```

Note the call to `skipBlanks()` at the end of the `while` loop. This subroutine must be executed before the computer returns to the test at the beginning of the loop. More generally, if the test in a `while` loop has a precondition, then you have to make sure that this precondition holds at the end of the `while` loop, before the computer jumps back to re-evaluate the test.

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 illegal.) 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`. You can try it in the applet at the end of this section. 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. As mentioned in the [previous section](#), the computer will get the value `Double.POSITIVE_INFINITY` when it does the computation. You might try this in the applet below to see what kind of output you get.

Here is the subroutine written out in Java:

```

static double readMeasurement() {

    // 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.

    double inches; // Total number of inches in user's measurement.

    double measurement; // One measurement,
                        // such as the 12 in "12 miles"
    String units;        // The units specified for the measurement,
                        // such as "miles"

    char ch; // Used to peek at next character in the user's input.

    inches = 0; // No inches have yet been read.

    skipBlanks();
    ch = TextIO.peek();

    /* As long as there is more input on the line, read a measurement and
       add the equivalent number of inches to the variable, inches.  If an
       error is detected during the loop, end the subroutine immediately
       by returning -1. */

    while (ch != '\n') {

        /* Get the next measurement and the units.  Before reading
           anything, make sure that a legal value is there to read. */

        if ( ! Character.isDigit(ch) ) {
            TextIO.putln(
                "Error:  Expected to find a number, but found " + ch);
            return -1;
        }
        measurement = TextIO.getDouble();

        skipBlanks();
        if (TextIO.peek() == '\n') {
            TextIO.putln(
                "Error:  Missing unit of measure at end of line.");
            return -1;
        }
        units = TextIO.getWord();
        units = units.toLowerCase();

        /* Convert the measurement to inches and add it to the total. */

        if (units.equals("inch")
            || units.equals("inches") || units.equals("in")) {
            inches += measurement;
        }
        else if (units.equals("foot"))
    }
}

```

```

        || units.equals("feet") || units.equals("ft")) {
    inches += measurement * 12;
}
else if (units.equals("yard")
        || units.equals("yards") || units.equals("yd")) {
    inches += measurement * 36;
}
else if (units.equals("mile")
        || units.equals("miles") || units.equals("mi")) {
    inches += measurement * 12 * 5280;
}
else {
    TextIO.putln("Error: \" + units
                + "\" is not a legal unit of measure.");
    return -1;
}

/* Look ahead to see whether the next thing on the line is
   the end-of-line. */

skipBlanks();
ch = TextIO.peek();

} // end while

return inches;

} // end readMeasurement()

```

The source code for the complete program can be found in the file [LengthConverter2.java](#). Here is an applet that simulates the program:

**(Applet "LengthConverter2Console" would be displayed here
if Java were available.)**

[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Section 9.3

Exceptions and the `try...catch` Statement

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. This could be done with an `if` statement:

```
if (i < 0 || i >= A.length) {
    ... // Do something to handle the out-of-range index, i
}
else {
    ... // Process the array element, A[i]
}
```

There are some problems with this approach. It is difficult and sometimes impossible to anticipate all the possible things that might go wrong. It's not always clear what to do when an error is detected. Furthermore, trying to anticipate all the possible problems can turn what would otherwise be a straightforward program into a messy tangle of `if` statements.

Java (like its cousin, C++) provides a neater, more structured alternative method for dealing with errors that can occur while a program is running. The method 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.)

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 the case of an applet, only the current operation -- such as the response to a button -- will be terminated. Parts of the applet might continue to function even when other parts are non-functional because of exceptions.) In many other programming languages, a crashed program will often 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.

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. The object thrown by an exception must be an instance of the standard class `java.lang.Throwable` or of one of its

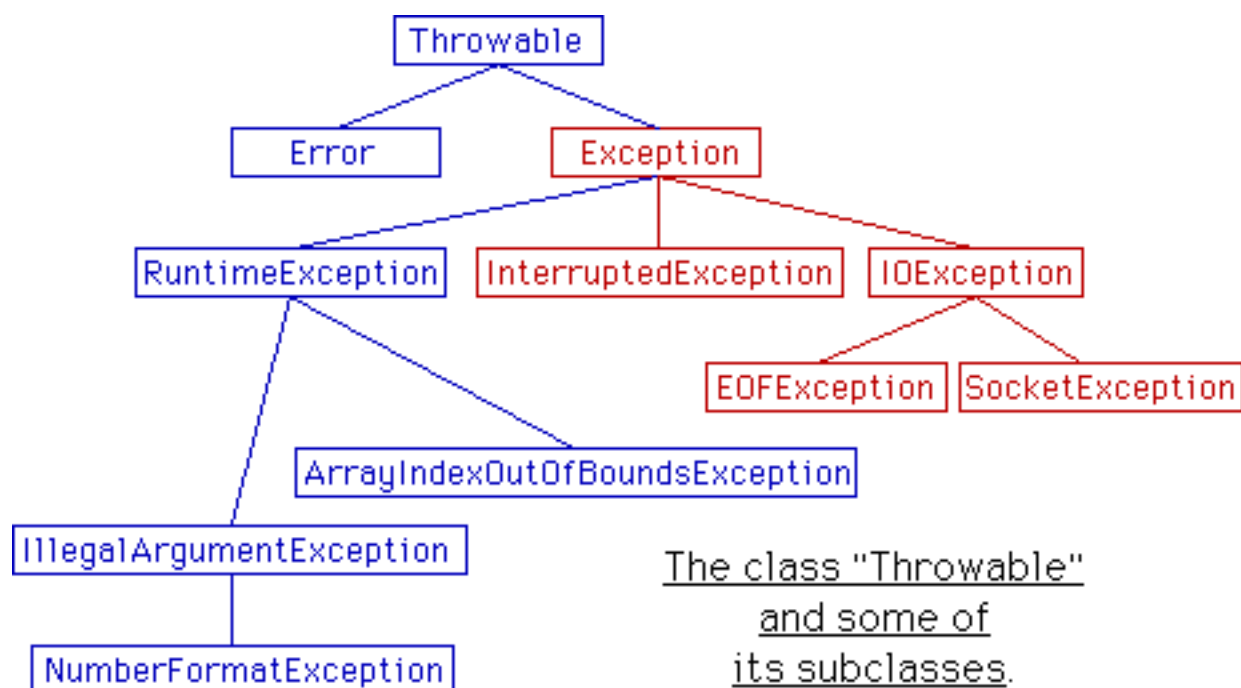
subclasses. In general, each different type of exception is represented by its own subclass of `Throwable`. `Throwable` has two direct subclasses, `Error` and `Exception`. These two subclasses in turn have many other predefined subclasses. In addition, a programmer can create new exception classes to represent new types of exceptions.

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. You should not try to catch and handle such errors. An example is the `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 usually best just to terminate the program, 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 such as: `ArithmeticException`, which occurs for example when there is an attempt to divide an integer by zero, `ArrayIndexOutOfBoundsException`, which occurs when an out-of-bounds index is used in an array, and `NullPointerException`, which occurs when there is an attempt to use a null reference in a context when an actual object reference is required. A `RuntimeException` generally indicates a bug in the program, which the programmer should fix. `RuntimeException`s 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.



To catch exceptions in a Java program, you need a `try` statement. The idea is that you tell the computer to "try" to execute some commands. If it succeeds, all well and good. But if an exception is thrown during the execution of those commands, you can catch the exception and handle it. 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 ( ArrayIndexOutOfBoundsException e ) {
    System.out.println("M is the wrong size to have a determinant.");
}
```

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 `ArrayIndexOutOfBoundsException` occurs, then the computer jumps immediately to the block of statements labeled "catch (`ArrayIndexOutOfBoundsException` e)". 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.

You might notice that there is another possible source of error in this try statement. If the value of the variable `M` is null, then a `NullPointerException` will be thrown when the attempt is made to reference the array. In the above try statement, `NullPointerException`s are not caught, so they will be processed in the ordinary way (by terminating the program, unless the exception is handled elsewhere). You could catch `NullPointerException`s by adding another 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:  " + );
    System.out.println( e.getMessage() );
}
```

This example shows how to use multiple catch clauses in one try block. It also shows what that little "e" is doing in the catch clauses. The `e` is actually a variable name. (You can use any name you like.) Recall that when an exception occurs, it is actually an object that is thrown. Before executing a catch clause, the computer sets this variable to refer to the exception object that is being caught. This object contains information about the exception. For example, an error message describing the exception can be retrieved using the object's `getMessage()` method, as is done in the above example. Another useful method in every exception object, `e`, is `e.printStackTrace()`. This method will print out the list of subroutines that were being executed when the exception was thrown. This information can help you to track down the part of your program that caused the error.

Note that both `ArrayIndexOutOfBoundsException` and `NullPointerException` are subclasses of `RuntimeException`. It's possible to catch all `RuntimeException`s 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 e ) {
    System.out.println("Sorry, an error has occurred.");
}
```

```

        e.printStackTrace();
    }

```

Since any object of type `ArrayIndexOutOfBoundsException` or of type `NullPointerException` is also of type `RuntimeException`, this will catch array index errors and null pointer errors as well as any other type of runtime exception. 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.

The example I've given here is not particularly realistic. You are not very likely to use exception-handling to guard against null pointers and bad array indices. This is a case where careful programming is better than exception handling: Just be sure that your program assigns a reasonable, non-null value to the array `M`. You would certainly resent it if the designers of Java forced you to set up a `try...catch` statement every time you wanted to use an array! This is why handling of potential `RuntimeExceptions` is not mandatory. There are just too many things that might go wrong! (This also shows that exception-handling does not solve the problem of program robustness. It just gives you a tool that will in many cases let you approach the problem in a more organized way.)

The syntax of a `try` statement is a little more complicated than I've indicated so far. The syntax can be described as

```

try {
    statements
}
optional-catch-clauses
optional-finally-clause

```

Note that this is a case where a block of statements, enclosed between `{` and `}`, is required. You need the `{` and `}` even if they enclose just one statement. The `try` statement can include zero or more `catch` clauses and, optionally, a `finally` clause. (The `try` statement must include either a `finally` clause or at least one `catch` clause.) The syntax for a `catch` clause is

```

catch ( exception-class-name variable-name ) {
    statements
}

```

and the syntax for a `finally` clause is

```

finally {
    statements
}

```

The semantics of the `finally` clause is that the block of statements in the `finally` clause is guaranteed to be executed as the last step in the execution of the `try` statement, whether or not any exception occurs and whether or not any exception that does occur is caught and handled. The `finally` clause is meant for doing essential cleanup that under no circumstances should be omitted.

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.

To throw an exception, use a `throw` statement. 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. (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? The answer is a little surprising: 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.)

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 that routine does not handle the exception, then it also is terminated and the routine that called it 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.

A subroutine that might generate an exception can announce this fact by adding the clause **"throws exception-class-name"** to the header of the routine. For example:

```
static double root(double A, double B, double C)
    throws IllegalArgumentException {
    // Returns the larger of the two roots of
    // the quadratic equation A*x*x + B*x + C = 0.
    // (Throws an exception if A == 0 or B*B-4*A*C < 0.)
    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 $A \neq 0$ and $B^2 - 4AC \geq 0$. 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.

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 `IllegalArgumentException`s 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.

On the other hand, suppose that some statement in a program can generate an exception 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. The second way is to declare that the subroutine that contains the statement can throw the exception. This is done by adding a `"throws"` clause to the subroutine heading. If the `throws` clause is used, then any other routine that calls the subroutine will be responsible for handling the exception. If you don't handle the possible exception in one of these two ways, it will be considered a syntax error, and the compiler will not accept your program.

Exception-handling is mandatory for any exception class that is not a subclass of either `Error` or `RuntimeException`. Exceptions that require mandatory handling 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. A robust program has to be prepared to handle such conditions. The design of Java makes it impossible for programmers to ignore such conditions.

Among the exceptions that require mandatory handling 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. The [next chapter](#) deals with input/output and uses exception-handling extensively.

[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Section 9.4

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.

Writing New Exception Classes

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, the new class will extend `RuntimeException` (or one of its subclasses) if the programmer does not want to require mandatory exception handling. To create a new exception class that 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) {
        // Constructor. 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 Section 5.5.](#)) 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 meaning of a string. A `ParseError` would indicate, presumably, that some string being processed by the program does not have the expected form.)

A `throw` statement can be used in a program to throw an error of type `ParseError`. The constructor for the `ParseError` object must specify an error message. For example:

```
throw new ParseError("Encountered an illegal negative number.");
```

or

```
throw new ParseError("The word '" + word
                    + "' is not a valid file name.");
```

If the `throw` statement does not occur in a `try` statement that catches the error, then the subroutine that

contains the `throw` statement must declare that it can throw a `ParseError`. It does this 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 exception handling for `ParseErrors` would not be mandatory.

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: Create a ShipDestroyed object
        // carrying an error message and 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.

Exceptions in Subroutines and Classes

The ability to throw exceptions is particularly useful in writing general-purpose subroutines and classes that are meant to be used in more than one program. In this case, the person writing the subroutine or class often has no reasonable way of handling the error, since that person has no way of knowing exactly how the subroutine 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 subroutine or uses the class needs to know that the error has occurred. In languages that do not support exceptions, the only alternative is to return some special value or to set the value of some variable to indicate that an error has occurred. For example, the `readMeasurement()` function in [Section 2](#) returns the value `-1` if the user's input is illegal. However, this only works if the main program bothers to test the return value. 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()` subroutine 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 earlier in this section. (Arguably, it might be more 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**:

```
static double readMeasurement() throws ParseError {

    // Reads the user's input measurement from one line of input.
    // Precondition:  The input line is not empty.
    // Postcondition: The measurement is converted to inches and
    //                returned. However, if the input is not legal,
    //                a ParseError is thrown.
    // Note:  The end-of-line is NOT read by this routine.

    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("\n" + units
                + "\n 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 Section 2, so I will not include an applet version of the program here. Internally, however, the programs are different, since LengthConverter3 uses exception-handling.

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, 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.

The programming languages C and C++ have a facility for adding **assertions** to a program. These assertions take the form `assert (condition)`, where **condition** is a boolean-valued expression. This condition expresses a precondition that must 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. Assertions of this form are not available in Java, but something similar can be done with exceptions. The Java equivalent of `assert (condition)` is:

```
if (condition == false)
    throw new IllegalArgumentException("Assertion Failed.");
```

Of course, you could use a better error message. And it would be better style to define a new exception class instead of using the standard class `IllegalArgumentException`.

Assertions are most useful during testing and debugging. Once you release your program, you don't really want it to crash. Still, many programs are released with a main program that says, essentially

```
try {
    .
    . // Run the program.
    .
}
catch (Exception e) {
    System.out.println("An unexpected internal error has occurred.");
    System.out.println("Please submit a bug report to the programmer.");
    System.out.println("Details of the error:");
    e.printStackTrace();
}
```

If a program contains a large number of assertions, they might slow the program down significantly. One advantage of assertions in C and C++ is that they can be "turned off." 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. 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. The nice part is that the source code doesn't have to be modified to produce the release version.

There is something similar that might work in Java, depending on how smart your compiler is. Suppose that you define a constant

```
static final boolean DEBUG = true;
```

and express your assertions as:

```
if (DEBUG == true && condition == false)
    throw new IllegalArgumentException("Assertion Failed.");
```

Since `DEBUG` is true, the value of "`DEBUG == true && condition == false`" is the same as the

value of **condition**, so this `if` statement still works as a test of the precondition. Now suppose you are finished debugging. Before you compile the release version of the program, change the definition of `DEBUG` to

```
static final boolean DEBUG = false;
```

Now, the value of "`DEBUG == true && condition == false`" has to be false, and a smart compiler can tell this at compilation time. Given that the condition in the `if` statement is known to be false, a smart compiler will not even bother to include the `if` statement in the compiled code, since it would not be executed in any case. So, the compiled code for the release version will be shorter and more efficient than the debugging version. And you only had to change one line in the source code!

End of Chapter 9

[[Next Chapter](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Programming Exercises For Chapter 9

THIS PAGE CONTAINS programming exercises based on material from [Chapter 9](#) of this [on-line Java textbook](#). Each exercise has a link to a discussion of one possible solution of that exercise.

Exercise 9.1: Write a program that uses the following subroutine, from [Section 3](#), to solve equations specified by the user.

```
static double root(double A, double B, double C)
    throws IllegalArgumentException {
    // Returns the larger of the two roots of
    // the quadratic equation A*x*x + B*x + C = 0.
    // (Throws an exception if A == 0 or B*B-4*A*C < 0.)
    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 9.2: As discussed in [Section 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 on your computer.) 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`.

`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 `BigInteger`s to represent the terms in the sequence. You can read the user's input into a `String` with the `TextIO.getln()` function. Use the input value to create the `BigInteger` object that represents the starting point of the $3N+1$ sequence. Don't forget to catch and handle the `NumberFormatException` that will occur if the user's input is not a legal integer! You should also check that the input number is greater than zero.

If the user's input is legal, print out the $3N+1$ sequence. Count the number of terms in the sequence, and print the count at the end of the sequence. Exit the program when the user inputs an empty line.

[See the solution!](#)

Exercise 9.3: 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. If that 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. Here is an applet that simulates my solution to this problem:

[See the solution!](#)

Exercise 9.4: The 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.

Finally, `func.getDefinition()` 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. Here is an applet that simulates my solution to this exercise, so that you can see how it works:

[See the solution!](#)

Exercise 9.5: This exercise uses the class `Expr`, which was described in Exercise 9.4. For this exercise, you should write an applet that can graph a function, $f(x)$, whose definition is entered by the user. The applet should have a text-input box where the user can enter an expression involving the variable `x`, such as x^2 or $\sin(x-3)/x$. This expression is the definition of the function. When the user presses return in the text input box, the applet 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 applet should display an error message. Otherwise, it should display a graph of the function. (Note: A `JTextField` generates an `ActionEvent` when the user presses return.)

The applet will need a `JPanel` for displaying the graph. To keep things simple, this panel should represent a fixed region in the xy -plane, defined by $-5 \leq x \leq 5$ and $-5 \leq y \leq 5$. 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 applet divides the interval $-5 \leq x \leq 5$ into 300 subintervals and uses the 301 endpoints of these subintervals for drawing the graph. Note that the function might be undefined at one of these x -values. In that case, you have to skip that point.

A point on the graph has the form (x, y) where y is obtained by evaluating the user's expression at the given value of x . You will have to convert these real numbers to the integer coordinates of the corresponding pixel on the canvas. The formulas for the conversion are:

```
a = (int)( (x + 5)/10 * width );
b = (int)( (5 - y)/10 * height );
```

where a and b are the horizontal and vertical coordinates of the pixel, and `width` and `height` are the width and height of the canvas.

Here is my solution to this exercise:

[See the solution!](#)

Quiz Questions For Chapter 9

THIS PAGE CONTAINS A SAMPLE quiz on material from [Chapter 9](#) of this [on-line Java textbook](#). You should be able to answer these questions after studying that chapter. Sample answers to all the quiz questions can be found [here](#).

Question 1: What does it mean to say that a program is *robust*?

Question 2: Why do programming languages require that variables be declared before they are used? What does this have to do with correctness and robustness?

Question 3: What is "Double.NaN"?

Question 4: What is a *precondition*? Give an example.

Question 5: Explain how preconditions can be used as an aid in writing correct programs.

Question 6: Java has a predefined class called `Throwable`. What does this class represent? Why does it exist?

Question 7: Write a subroutine that prints out a $3N+1$ sequence starting from a given integer, N . The starting value should be a parameter to the subroutine. 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 8: Some classes of exceptions 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?

[[Answers](#) | [Chapter Index](#) | [Main Index](#)]

Chapter 10

Advanced Input/Output

COMPUTER PROGRAMS ARE ONLY USEFUL if they interact with the rest of the world in some way. This interaction is referred to as input/output, or I/O. Up until now, the only type of interaction that has been covered in this textbook is 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. In this chapter, we'll look at others, including files and network connections. In Java, input/output involving files and networks is based on **streams**, which are objects that support the same sort of I/O commands that you have already used to communicate with the user in a command-line interface. In fact, standard output (`System.out`) and standard input (`System.in`) are examples of streams.

Working with files and networks requires familiarity with exceptions, which were introduced in the [previous chapter](#). Many of the subroutines that are used can throw exceptions that require mandatory exception handling. This generally means calling the subroutine in a `try...catch` statement that can deal with the exception if one occurs.

Contents of Chapter 10:

- Section 1: [Streams, Readers, and Writers](#)
- Section 2: [Files](#)
- Section 3: [Programming with Files](#)
- Section 4: [Networking](#)
- Section 5: [Threads and Network Programming](#)
- [Programming Exercises](#)
- [Quiz on this Chapter](#)

[[First Section](#) | [Next Chapter](#) | [Previous Chapter](#) | [Main Index](#)]

Section 10.1

Streams, Readers, and Writers

WITHOUT THE ABILITY TO INTERACT WITH the rest of the world, a program would be useless. The interaction of a program with the rest of the world is referred to as **input/output** or I/O. Historically, one of the hardest parts of programming language design has been coming up with good facilities for doing input and output. A computer can be connected to many different types of input and output devices. If a programming language had to deal with each type of device as a special case, the complexity would be overwhelming. One of the major achievements in the history of programming has been to come up with good abstractions for representing I/O devices. In Java, the I/O abstractions are called **streams**. This section is an introduction to streams, but it is not meant to cover them in full detail. See the official Java documentation for more information.

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 data. Machine-formatted data is represented in the same way that data is represented inside the computer, that is, as strings of zeros and ones. Human-readable data is in the form of characters. When you read a number such as 3.141592654, you are reading a sequence of characters and interpreting them as a number. The same number would be represented in the computer as a bit-string that you would find unrecognizable.

To deal with the two broad categories of data representation, Java has two broad categories of streams: **byte streams** for machine-formatted data and **character streams** for human-readable data. There are many predefined classes that represent streams of each type.

Every object that **outputs data to a byte stream belongs to one of the subclasses of the abstract class `OutputStream`**. Objects that read data from a byte stream belong to subclasses of `InputStream`. **If you write numbers to an `OutputStream`, you won't be able to read the resulting data yourself. But the data can be read back into the computer with an `InputStream`. The writing and reading of the data will be very efficient, since there is no translation involved: the bits that are used to represent the data inside the computer are simply copied to and from the streams.**

For reading and writing human-readable character data, the main classes are `Reader` and `Writer`. All character stream classes are subclasses of one of these. If a number is to be written to a `Writer` stream, the computer must translate it into a human-readable sequence of characters that represents that number. Reading a number from a `Reader` stream into a numeric variable also involves a translation, from a character sequence into the appropriate bit string. (Even if the data you are working with consists of characters in the first place, such as words from a text editor, there might still be some translation. Characters are stored in the computer as 16-bit Unicode values. For people who use Western alphabets, character data is generally stored in files in ASCII code, which uses only 8 bits per character. The `Reader` and `Writer` classes take care of this translation, and can also handle non-western alphabets in countries that use them.)

It's usually easy to decide whether to use byte streams or character streams. If you want the data to be human-readable, use character streams. Otherwise, use byte streams. I should note that Java 1.0 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, as of Java 1.1, you should use `Readers` and `Writers` rather than `InputStreams` and `OutputStreams` when working with character data.

The standard stream classes discussed in this section are defined in the package `java.io`, along with several supporting classes. You must import the classes from this package if you want to use them in your program. That means putting the directive `"import java.io.*;"` at the beginning of your source file. Streams are not used in Java's graphical user interface, which has its own form of I/O. But they are

necessary for working with files and for doing communication over a network. They can be also used for communication between two concurrently running threads, and there are stream classes for reading and writing data stored in the computer's memory.

The beauty of the stream abstraction is that it is as easy to write data to a file or to send data over a network as it is to print information on the screen.

The basic I/O classes `Reader`, `Writer`, `InputStream`, and `OutputStream` provide only very primitive I/O operations. For example, the `InputStream` class declares the instance method

```
public int read() throws IOException
```

for reading one byte of data (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 `IOException` is thrown. Since `IOException` is an exception class that requires mandatory exception-handling, this means that you can't use the `read()` method except inside a `try` statement or in a subroutine that is itself declared with a "throws `IOException`" clause. (Exceptions and `try...catch` statements were covered in [Chapter 9](#).)

The `InputStream` class also defines methods for reading several bytes of data in one step into an array of bytes. However, `InputStream` provides no convenient methods for reading other types of data, such as `int` or `double`, from a stream. This is not a problem because you'll never use an object of type `InputStream` itself. Instead, you'll use subclasses of `InputStream` that add more convenient input methods to `InputStream`'s rather primitive capabilities. Similarly, the `OutputStream` class defines a primitive output method for writing one byte of data to an output stream, the method

```
public void write(int b) throws IOException
```

but again, in practice, you will almost always use higher-level output operations defined in some subclass of `OutputStream`.

The `Reader` and `Writer` classes provide very similar low-level read and write operations. But in these character-oriented classes, the I/O operations read and write `char` values rather than bytes. In practice, you will use sub-classes of `Reader` and `Writer`, as discussed below.

One of the neat things about Java's I/O package is that it lets you add capabilities to a stream by "wrapping" it in another stream object that provides those capabilities. The wrapper object is also a stream, so you can read from or write to it -- but you can do so using fancier operations than those available for basic streams.

For example, `PrintWriter` is a subclass of `Writer` that provides convenient methods for outputting human-readable character representations of all of Java's basic data types. If you have an object belonging to the `Writer` class, or any of its subclasses, and you would like to use `PrintWriter` methods to output data to that `Writer`, all you have to do is wrap the `Writer` in a `PrintWriter` object. You do this by constructing a new `PrintWriter` object, using the `Writer` as input to the constructor. For example, if `charSink` is of type `Writer`, then you could say

```
PrintWriter printableCharSink = new PrintWriter(charSink);
```

When you output data to `printableCharSink`, using `PrintWriter`'s advanced data output methods, that data will go to exactly the same place as data written directly to `charSink`. You've just provided a better interface to the same output stream. For example, this allows you to use `PrintWriter` methods to send data to a file or over a network connection.

For the record, the output methods of the `PrintWriter` class include:

```
public void print(String s)    // Methods for outputting
```

```

public void print(char c)      // standard data types
public void print(int i)      // to the stream, in
public void print(long l)     // human-readable form.
public void print(float f)
public void print(double d)
public void print(boolean b)

public void println() // Output a carriage return to the stream.

public void println(String s) // These methods are identical
public void println(char c)  // to the previous set,
public void println(int i)   // except that the output
public void println(long l)  // value is followed by
public void println(float f) // a carriage return.
public void println(double d)
public void println(boolean b)

```

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 every time you use a `PrintWriter` method.

When you use `PrintWriter` methods to output data to a stream, the data is converted into the sequence of characters that represents the data in human-readable form. Suppose you want to output the data in byte-oriented, machine-formatted form? The `java.io` package includes a byte-stream class, `DataOutputStream` that can be used for writing data values to streams in internal, binary-number format. `DataOutputStream` bears the same relationship to `OutputStream` that `PrintWriter` bears to `Writer`. That is, whereas `OutputStream` only has methods for outputting bytes, `DataOutputStream` has methods `writeDouble(double x)` for outputting values of type `double`, `writeInt(int x)` for outputting values of type `int`, and so on. Furthermore, you can wrap any `OutputStream` in a `DataOutputStream` so that you can use the higher level output methods on it. For example, if `byteSink` is of type `OutputStream`, you could say

```
DataOutputStream dataSink = new DataOutputStream(byteSink);
```

to wrap `byteSink` in a `DataOutputStream`, `dataSink`.

For input of machine-readable data, such as that created by writing to a `DataOutputStream`, `java.io` provides the class `DataInputStream`. You can wrap any `InputStream` in a `DataInputStream` object to provide it with the ability to read data of various types from the byte-stream. The methods in the `DataInputStream` for reading binary data are called `readDouble()`, `readInt()`, and so on. Data written by a `DataOutputStream` is guaranteed to be in a format that can be read by a `DataInputStream`. This is true even if the data stream is created on one type of computer and read on another type of computer. The cross-platform compatibility of binary data is a major aspect of Java's platform independence.

Still, the fact remains that 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`.

Fortunately, Java's object-oriented nature makes it possible to write such a class and then use it in exactly the same way as if it were a standard part of the language.

Following this model, I have written a class called `TextReader` that allows convenient input of data that was written in human-readable character format. The [source code](#) for this class is available if you want to read it. A `TextReader` can be used as a wrapper for an existing input stream. The constructor

```
public TextReader(Reader dataSource)
```

creates an object that can be used to read data from the given `Reader`, `dataSource`, using the convenient input methods of the `TextReader` class. The methods in my `TextReader` class are similar to the static input methods in my `TextIO` class, except that `TextReaders` can be used to read from any input stream, whereas `TextIO` can only be used to read from the standard input stream, `System.in`. Instance methods in the `TextReader` class include:

```
public char peek()    // Look at the next character in the stream,
                    //    without removing it from the stream.  If
                    //    the characters in the stream have all
                    //    been read, then the character '\0' is
                    //    returned.  If the next character in the
                    //    stream is a carriage return, then a '\n'
                    //    is returned.

public char getAnyChar() // Reads the next character from the
                    //    stream.  It can be a whitespace
                    //    character.  If all the characters
                    //    in the stream have been read, an
                    //    error occurs.

public void skipWhiteSpace() // Read and discard whitespace
                    //    characters (space, return, tab),
                    //    until a non-whitespace character
                    //    is seen.

public boolean eoln()    // Discards spaces or tabs in the stream,
                    //    then tests whether the next char is
                    //    the end of the current line (or the
                    //    end of the data in the stream).

public boolean eof()    // Discards any whitespace characters, then
                    //    returns true if all the characters
                    //    in the stream have been read.

public char getChar()    // These routines read values of the
public byte getByte()    // specified types.  In each case,
public short getShort() // the computer skips any whitespace
public int getInt()      // characters before trying to read a
public long getLong()    // value of the specified type.
public float getFloat()  // An error occurs if a value of the
public double getDouble() // correct type is not found.  For
public String getWord()  // the getWord() routine, a word is
public boolean getBoolean() // considered to be any string of
                    // non-blank characters.  For
                    // getBoolean(), the input can be any
                    // of the strings "true", "false", "t",
                    // "f", "yes", "no", "y", "n", "1",
                    // or "0", ignoring case.
```

```

public String getAlpha()    // This is similar to getWord(), except
                           // that it returns a string consisting
                           // of letters only. It is also special
                           // in that it skips over any non-letters
                           // before reading a word, rather than
                           // just skipping over white space.

public String getln();      // Reads characters up to the end of the
                           // current line of input. Then reads
                           // and discards the carriage return.
                           // Note that this routine does NOT skip
                           // leading whitespace characters, and
                           // that the value returned might be the
                           // empty string.

public char getlnChar();    // These routines are provided as a
public byte getlnByte();    // convenience. They are equivalent
public short getlnShort();  // to the above routines, except that
public int getlnInt();      // after successfully reading a value
public long getlnLong();    // of the specified type, the computer
public float getlnFloat();  // reads and discards any remaining
public double getlnDouble(); // characters on the same line.
public String getlnString();
public boolean getlnBoolean();
public String getlnAlpha();

```

For convenience, I also make it possible to wrap an `InputStream` in a `TextReader` object, in the same way that it is possible to wrap a `Reader` object in a `TextReader`. For example, since `System.in` is of type `InputStream`, you could say:

```
TextReader in = new TextReader(System.in);
```

The `TextReader`, `in`, could then be used in much the same way as the `TextIO` class. For example, you could use `in.getInt()` to read an integer from standard input or use `in.getBoolean()` to read a boolean value. The only difference would be that the `TextReader` does not handle errors in the input in the same way as `TextIO`. In an exactly symmetrical way, you can wrap an `OutputStream` in a `PrintWriter` if you want to write character data to the stream.

There remains the question of what happens when an error occurs while one of the input routines in the `TextReader` class is being executed. Whoever designed the `PrintWriter` class decided not to throw exceptions when errors occur. When I designed `TextReader`, I decided to give you a choice. By default, a routine that encounters an error will throw an exception belonging to the class `TextReader.Error`. This is a static nested class declared inside the `TextReader` class. (For information on nested classes, see [Section 5.6.](#)) `TextReader.Error` is a subclass of the `RuntimeException` class. You can catch the error in a `try...catch` statement and handle it, if you want. Recall that the compiler does not force you to use `try` and `catch` to deal with `RuntimeExceptions`. However, if one occurs and is not caught, it will crash your program. If you prefer not to work with exceptions at all, you can turn off this behavior by calling the `TextReader` instance method

```
public void checkIO(boolean throwExceptions)
```

with its parameter set to `false`. In that case, when an error occurs during input, no exception will be thrown. Instead, the value of an internal error flag will be set, and the program will continue. If you use this option, it is your responsibility to check for errors after each input operation. You can do this with the instance method

```
public boolean checkError()
```

This method returns `true` if the most recent input operation on the `TextReader` produced an error, and it returns `false` if that operation completed successfully. It is probably easier to write robust programs by catching and handling exceptions than by continually checking for possible errors. With both options available, you can experiment with both styles of error-handling and see which one you prefer.

The classes `PrintWriter`, `TextReader`, `DataInputStream`, and `DataOutputStream` allow you to easily input and output all of Java's primitive data types. But what happens when you want to read and write objects? Traditionally, you would have to come up with some way of encoding your object as a sequence of data values belonging to the primitive types, which can then be output as bytes or characters. This is called **serializing** the object. On input, you have to read the serialized data and somehow reconstitute a copy of the original object. For complex objects, this can all be a major chore. However, you can get Java to do all the work for you by using the classes `ObjectInputStream` and `ObjectOutputStream`. These are subclasses of `InputStream` and `OutputStream` that can be used for writing and reading serialized objects.

`ObjectInputStream` and `ObjectOutputStream` are wrapper classes that can be wrapped around arbitrary `InputStreams` and `OutputStreams`. This makes it possible to do object input and output on any byte-stream. The methods for object I/O are `readObject()`, in `ObjectInputStream`, and `writeObject(Object obj)`, in `ObjectOutputStream`. Both of these methods can throw `IOExceptions`. Note that `readObject()` returns a value of type `Object`, which generally has to be type-cast to a more useful type.

`ObjectInputStream` and `ObjectOutputStream` only work with objects that implement an interface named `Serializable`. Furthermore, all of the instance variables in the object must be serializable. However, there is little work involved in making an object serializable, since the `Serializable` interface does not declare any methods. It exists only as a marker for the compiler, to tell it that the object is meant to be writable and readable. You only need to add the words "implements `Serializable`" to your class definitions. Many of Java's standard classes are already declared to be serializable, including all the component classes in Swing and in the AWT. This means, in particular, that GUI components can be written to `ObjectOutputStreams` and read from `ObjectInputStreams`.

[[Next Section](#) | [Previous Chapter](#) | [Chapter Index](#) | [Main Index](#)]

Section 10.2

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 floppy disk, on a CD-ROM, or on some other type of storage device. Files are organized into **directories** (sometimes called "folders"). A directory can hold other directories, as well as files. Both directories and files have names that are used to identify them.

Programs can read data from existing files. They can create new files and can write data to files. In Java, such input and output is done using streams. Human-readable character data is read from a file using an object belonging to the class `FileReader`, which is a subclass of `Reader`. Similarly, data is written to a file in human-readable format through an object of type `FileWriter`, a subclass of `Writer`. For files that store data in machine format, the appropriate I/O classes are `FileInputStream` and `FileOutputStream`. In this section, I will only discuss character-oriented file I/O using the `FileReader` and `FileWriter` classes. However, `FileInputStream` and `FileOutputStream` are used in an exactly parallel fashion. All these classes are defined in the `java.io` package.

It's worth noting right at the start that applets which are downloaded over a network connection are generally not allowed to access files. This is a security consideration. You can download and run an applet just by visiting a Web page with your browser. If downloaded applets had access to the files on your computer, it would be easy to write an applet that would destroy all the data on a computer that downloads it. To prevent such possibilities, there are a number of things that downloaded applets are not allowed to do. Accessing files is one of those forbidden things. Standalone programs written in Java, however, have the same access to your files as any other program. When you write a standalone Java application, you can use all the file operations described in this section.

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. This exception type requires mandatory exception handling, so you have to call the constructor in a `try` statement (or inside a routine that is declared to throw `FileNotFoundException`). 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 `IOException`s 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 `TextReader` object or in some other wrapper class. (The

`TextReader` class is not a standard part of Java; it is described in the [previous section](#).) To create a `TextReader` for reading from a file named `data.dat`, you could say:

```
TextReader data;

try {
    data = new TextReader(new FileReader("data.dat"));
}
catch (FileNotFoundException e) {
    ... // handle the exception
}
```

Once you have a `TextReader` named `data`, you can read from it using such methods as `data.getInt()` and `data.getWord()`, exactly as you would from any other `TextReader`.

Working with output files is no more difficult than this. You simply create an object belonging to the class `FileWriter`. You will probably want to wrap this output stream in an object of type `PrintWriter`. For example, suppose you want to write data to a file named `result.dat`. Since the constructor for `FileWriter` can throw an exception of type `IOException`, you should use a `try` statement:

```
PrintWriter result;

try {
    result = new PrintWriter(new FileWriter("result.dat"));
}
catch (IOException e) {
    ... // handle the exception
}
```

If no file named `result.dat` exists, a new file will be created. If the file already exists, then the current contents of the file will be erased and replaced with the data that your program writes to the file. An `IOException` might occur if, for example, you are trying to create a file on a disk that is "write-protected," meaning that it cannot be modified.

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. (If you forget to do this, the file will ordinarily be closed automatically when the program terminates or when the file stream object is garbage collected, but it's best to close a file as soon as you are done with it.) You can close a file by calling the `close()` method of the associated stream. Once a file has been closed, it is no longer possible to read data from it or write data to it, unless you open it again as a new stream. (Note that for most stream classes, the `close()` method can throw an `IOException`, which must be handled; however, both `PrintWriter` and `TextReader` override this method so that it cannot throw such exceptions.)

As a complete example, here is a program that will read numbers from a file named `data.dat`, and will then write out the same numbers in reverse order to another file named `result.dat`. It is assumed that `data.dat` contains only one number on each line, and that there are no more than 1000 numbers altogether. Exception-handling is used to check for problems along the way. Although the application is not a particularly useful one, this program demonstrates the basics of working with files. (By the way, at the end of this program, you'll find our first example of a `finally` clause in a `try` statement. When the computer executes a `try` statement, the commands in its `finally` clause are guaranteed to be executed, no matter what.)

```
import java.io.*;
// (The TextReader class must also be available to this program.)

public class ReverseFile {
```

```

public static void main(String[] args) {

    TextReader data;        // Character input stream for reading data.
    PrintWriter result;     // Character output stream for writing data.

    double[] number = new double[1000]; // An array to hold all
                                         // the numbers that are
                                         // read from the file.

    int numberCt; // Number of items actually stored in the array.

    try { // Create the input stream.
        data = new TextReader(new FileReader("data.dat"));
    }
    catch (FileNotFoundException e) {
        System.out.println("Can't find file data.dat!");
        return; // End the program by returning from main().
    }

    try { // Create the output stream.
        result = new PrintWriter(new FileWriter("result.dat"));
    }
    catch (IOException e) {
        System.out.println("Can't open file result.dat!");
        System.out.println(e.toString());
        data.close(); // Close the input file.
        return;       // End the program.
    }

    try {

        // Read the data from the input file.

        numberCt = 0;
        while (data.eof() == false) { // Read until end-of-file.
            number[numberCt] = data.getLnDouble();
            numberCt++;
        }

        // Output the numbers in reverse order.

        for (int i = numberCt-1; i >= 0; i--)
            result.println(number[i]);

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

    }
    catch (TextReader.Error e) {
        // Some problem reading the data from the input file.
        System.out.println("Input Error: " + e.getMessage());
    }
    catch (IndexOutOfBoundsException e) {
        // Must have tried to put too many numbers in the array.
        System.out.println("Too many numbers in data file.");
        System.out.println("Processing has been aborted.");
    }
}

```

```

        finally {
            // Finish by closing the files,
            //      whatever else may have happened.
            data.close();
            result.close();
        }

    } // end of main()

} // end of class

```

File Names, Directories, and the File Class

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** (or "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 its 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 quite a bit 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 the UNIX operating system. 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 DOS or Windows computer.
- `Hard Drive:java:examples:data.dat` -- Assuming that "Hard Drive" is the name of a disk drive, this would be an absolute path name on a computer using Macintosh OS 9.
- `examples/data.dat` -- a relative path name under UNIX. "Examples" is the name of a directory that is contained within the current directory, and `data.data` is a file in that directory. The corresponding relative path names for Windows and Macintosh would be `examples\data.dat` and `examples:data.dat`.

Similarly, the rules for determining which directory is the current directory are different for different types of computers. It's reasonably safe to say, though, that if you stick to using simple file names only, and if the files are stored in the same directory with the program that will use them, then you will be OK.

To avoid some of the problems caused by differences between platforms, Java has the class `java.io.File`. An object belonging to this class represents a file. More precisely, an object of type `File` represents a file *name* rather than a file as such. **The file to which the name refers might or might not exist. Directories are treated in the same way as files, so a `File` object can represent a directory just as easily as it can represent a file.**

A `File` object has a constructor `new File(String)` that creates a `File` object from a path name. The name can be a simple name, a relative path, or an absolute path. For example, `new File("data.dat")` creates a `File` object that refers to a file named `data.dat`, in the current directory. Another constructor,

`new File(File,String)`, has two parameters. The first is a `File` object that refers to the directory that contains the file. The second is the name of the file. Later in this section, we'll look at a convenient way of letting the user specify a `File` in a GUI program.

`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 could use this method if you wanted to avoid overwriting the contents of an existing file when you create a new `FileWriter`.

`file.isDirectory()` -- This boolean-valued function returns `true` if the `File` object refers to a directory. It returns `false` if it refers to a regular file or if no file with the given name exists.

`file.delete()` -- Deletes the file, if it exists.

`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`.

Here, for example, is a program that will list the names of all the files in a directory specified by the user:

```
import java.io.File;

public class DirectoryList {

    /* 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 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.

        TextIO.put("Enter a directory name: ");
        directoryName = TextIO.getln().trim();
        directory = new File(directoryName);

        if (directory.isDirectory() == false) {
            if (directory.exists() == false)
                TextIO.putln("There is no such directory!");
            else
                TextIO.putln("That file is not a directory.");
        }
        else {
            files = directory.list();
            TextIO.putln("Files in directory \"" + directory + "\"");
            for (int i = 0; i < files.length; i++)
                TextIO.putln("    " + 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)`. If you want to use a `TextReader` to read from the file, you might use:

```
TextReader data;

try {
    data = new TextReader( new FileReader(file) );
}
catch (FileNotFoundException e) {
    ... // handle the exception
}
```

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 special window that a program can open when it wants the user to select a file for input or output. Swing includes a platform-independent technique for using file dialog boxes in the form of a class called `JFileChooser`. This class is part of the package `javax.swing`. We looked at some basic dialog boxes in [Section 7.7](#). File dialog boxes are similar to those, but are a little more complicated to use.

A file dialog box shows the user a list of files and sub-directories in some directory, and makes it easy for the user to specify a file in that directory. The user can also navigate easily from one directory to another. The most common constructors for `JFileChooser` specify the directory that is selected when the dialog box first appears:

```
new JFileChooser( File startDirectory )

new JFileChooser( String pathToStartDirectory )
```

There is also a constructor with no arguments that will set the user's home directory to be the starting directory in the dialog box. (The constructor call `new JFileChooser(". ")` produces a dialog box that has the current directory as its starting directory. This is true since `"."` is a special path name that refers to the current directory, at least on Windows and UNIX systems.)

Constructing a `JFileChooser` object does not make the dialog box appear on the screen. You have to call a methods in the object to do that. There are two different methods that can be used because there are two types of file dialog: An **open file dialog** allows the user to specify an existing file to be opened for reading data into the program; a **save file dialog** lets the user specify a file, which might or might not already exist, to be opened for writing data from the program. File dialogs of these two types are opened using the `showOpenDialog` and `showSaveDialog` methods.

A file dialog box always has a **parent**, another component which is associated with the dialog box. The parent is specified as a parameter to the `showOpenDialog` or `showSaveDialog` methods. The parent is a GUI component, and can usually be specified as `"this"`. (The parameter can be null, in which case an invisible component is used as the parent.) Both `showOpenDialog` and `showSaveDialog` have a return value, which will be one of the constants `JFileChooser.CANCEL_OPTION`, `JFileChooser.ERROR_OPTION` or `JFileChooser.APPROVE_OPTION`. If the return value is

`JFileChooser.APPROVE_OPTION`, then the user has selected a file. If the return value is something else, then the user did not select a file. The user might have clicked a "Cancel" button, for example. You should always check the return value, to make sure that the user has, in fact, selected a file. If that is the case, then you can find out which file was selected by calling the `JFileChooser`'s `getFile()` method, which returns an object of type `File` that represents the selected file.

Putting all this together, typical code for using a `JFileChooser` to read character data from a file looks like this:

```
JFileChooser fileDialog = new JFileChooser(".");
int option = fileDialog.showOpenDialog(this);
if (option == JFileChooser.APPROVE_OPTION) {
    File selectedFile = fileDialog.getFile();
    try {
        TextReader data = new TextReader(new FileReader(selectedFile));
    }
    catch (FileNotFoundException e) {
        // Handle the error.
    }
    .
    . // Read data from the file.
    .
}
```

The first line creates a new `JFileChooser` object in which the current directory is initially selected. The second line shows the file dialog box on the screen and waits for the user to select a file or close the dialog box in some other way. The third line tests whether the user has actually selected a file. Only in that case do we proceed to get the selected file, open it, and use it. Writing data to a file would be similar, but `showSaveDialog` would replace `showOpenDialog`.

There is nothing to stop you, by the way, from using the same `JFileChooser` object over and over. This would have the advantage that the selected directory would be remembered from one use to the next.

It's common to do some configuration of a `JFileChooser` before calling `showOpenDialog` or `showSaveDialog`. For example, the instance method `setDialogTitle(String)` can be used to specify a title to appear at the top of the window. And `setSelectedFile(File)` can be used to set the file that is selected in the dialog box when it appears. This can be used to provide a default file choice for the user.

We'll look at some more complete examples of using files and file dialogs in the next section.

[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Section 10.3

Programming With Files

IN THIS SECTION, we look at several programming examples that work with files. The techniques that we need were introduced in [Section 1](#) and [Section 2](#).

The first example is a program that makes a list of all the words that occur in a specified file. The user is asked to type in the name of the file. The list of words is written to another file, which the user also specifies. A word here just means a sequence of letters. The list of words will be output in alphabetical order, with no repetitions. All the words will be converted into lower case, so that, for example, "The" and "the" will count as the same word.

Since we want to output the words in alphabetical order, we can't just output the words as they are read from the input file. We can store the words in an array, but since there is no way to tell in advance how many words will be found in the file, we need a "dynamic array" which can grow as large as necessary. Techniques for working with dynamic arrays were discussed in [Section 8.3](#). The data is represented in the program by two static variables:

```
static String[] words;    // An array that holds the words.
static int wordCount;    // The number of words currently
                        // stored in the array.
```

The program starts with an empty array. Every time a word is read from the file, it is inserted into the array (if it is not already there). The array is kept at all times in alphabetical order, so the new word has to be inserted into its proper position in that order. The insertion is done by the following subroutine:

```
static void insertWord(String w) {

    int pos = 0;    // This will be the position in the array
                  // where the word w belongs.

    w = w.toLowerCase();    // Convert word to lower case.

    /* Find the position in the array where w belongs, after all the
       words that precede w alphabetically. If a copy of w already
       occupies that position, then it is not necessary to insert
       w, so return immediately. */

    while (pos < wordCount && words[pos].compareTo(w) < 0)
        pos++;
    if (pos < wordCount && words[pos].equals(w))
        return;

    /* If the array is full, make a new array that is twice as
       big, copy all the words from the old array to the new,
       and set the variable, words, to refer to the new array. */

    if (wordCount == words.length) {
        String[] newWords = new String[words.length*2];
        System.arraycopy(words, 0, newWords, 0, wordCount);
        words = newWords;
    }

    /* Put w into its correct position in the array. Move any
```



```

        words that come after w up one space in the array to
        make room for w. */

    for (int i = wordCount; i > pos; i--)
        words[i] = words[i-1];
    words[pos] = w;
    wordCount++;

} // end insertWord()

```

This subroutine is called by the `main()` routine of the program to process each word that it reads from the file. If we ignore the possibility of errors, an algorithm for the program is

```

Get the file names from the user
Create a TextReader for reading from the input file
Create a PrintWriter for writing to the output file
while there are more words in the input file:
    Read a word from the input file
    Insert the word into the words array
For i from 0 to wordCount - 1:
    Write words[i] to the output file

```

Most of these steps can generate `IOExceptions`, and so they must be done inside `try...catch` statements. In this case, we'll just print an error message and terminate the program when an error occurs.

If `in` is the name of the `TextReader` that is being used to read from the input file, we can read a word from the file with the function `in.getAlpha()`. But testing whether there are any more words in the file is a little tricky. The function `in.eof()` will check whether there are any more non-whitespace characters in the file, but that's not the same as checking whether there are more words. It might be that all the remaining non-whitespace characters are non-letters. In that case, trying to read a word will generate an error, even though `in.eof()` is false. The fix for this is to skip all non-letter characters before testing `in.eof()`. The function `in.peek()` allows us to look ahead at the next character without reading it, to check whether it is a letter. With this in mind, the while loop in the algorithm can be written in Java as:

```

while (true) {
    while ( ! in.eof() && ! Character.isLetter(in.peek()) )
        in.getAnyChar(); // Read the non-letter character.
    if ( in.eof() ) // End if there is nothing more to read.
        break;
    insertWord( in.getAlpha() );
}

```

With error-checking added, the complete `main()` routine is as follows. If you want to see the program as a whole, you'll find the source code in the file [WordList.java](#).

```

public static void main(String[] args) {

    TextReader in;    // A stream for reading from the input file.
    PrintWriter out;  // A stream for writing to the output file.

    String inputFileName; // Input file name, specified by the user.
    String outputFileName; // Output file name, specified by the user.

    words = new String[10]; // Start with space for 10 words.
    wordCount = 0;          // Currently, there are no words in array.

    /* Get the input file name from the user and try to create the

```

```

        input stream.  If there is a FileNotFoundException, print
        a message and terminate the program. */

    TextIO.put("Input file name? ");
    inputFileName = TextIO.getln().trim();
    try {
        in = new TextReader(new FileReader(inputFileName));
    }
    catch (FileNotFoundException e) {
        TextIO.putln("Can't find file \"" + inputFileName + "\".");
        return; // Returning from main() ends the program.
    }

    /* Get the output file name from the user and try to create the
       output stream.  If there is an IOException, print a message
       and terminate the program. */

    TextIO.put("Output file name? ");
    outputFileName = TextIO.getln().trim();
    try {
        out = new PrintWriter(new FileWriter(outputFileName));
    }
    catch (IOException e) {
        TextIO.putln("Can't open file \"" +
                    outputFileName + "\" for output.");
        TextIO.putln(e.toString());
        return;
    }

    /* Read all the words from the input stream and insert them into
       the array of words.  Reading from a TextReader can result in
       an error of type TextReader.Error.  If one occurs, print an
       error message and terminate the program. */

    try {
        while (true) {
            // Skip past any non-letters in the input stream.  If
            // end-of-stream has been reached, end the loop.
            // Otherwise, read a word and insert it into the
            // array of words.
            while ( ! in.eof() && ! Character.isLetter(in.peek()) )
                in.getAnyChar();
            if (in.eof())
                break;
            insertWord(in.getAlpha());
        }
    }
    catch (TextReader.Error e) {
        TextIO.putln("An error occurred while reading from input file.");
        TextIO.putln(e.toString());
        return;
    }

    /* Write all the words from the list to the output stream. */

    for (int i = 0; i < wordCount; i++)

```

```

        out.println(words[i]);

    /* Finish up by checking for an error on the output stream and
       printing either a warning message or a message that the words
       have been output to the output file.  The PrintWriter class
       does not throw an exception when an error occurs, so we have
       to check for errors by calling the checkError() method. */

    if (out.checkError() == true) {
        TextIO.putln("Some error occurred while writing output.");
        TextIO.putln("Output might be incomplete or invalid.");
    }
    else {
        TextIO.putln(wordCount + " words from \"" + inputFileName +
                     "\" output to \"" + outputFileName + "\".");
    }
} // end main()

```

Making a copy of a file is a pretty common operation, and most operating systems already have a command for doing so. However, it is still instructive to look at a Java program that does the same thing. Many file operations are similar to copying a file, except that the data from the input file is processed in some way before it is written to the output file. All such operations can be done by programs with the same general form.

Since the program should be able to copy any file, we can't assume that the data in the file is in human-readable form. So, we have to use `InputStream` and `OutputStream` to operate on the file rather than `Reader` and `Writer`. The program simply copies all the data from the `InputStream` to the `OutputStream`, one byte at a time. If `source` is the variable that refers to the `InputStream`, then the function `source.read()` can be used to read one byte. This function returns the value `-1` when all the bytes in the input file have been read. Similarly, if `copy` refers to the `OutputStream`, then `copy.write(b)` writes one byte to the output file. So, the heart of the program is a simple while loop. (As usual, the I/O operations can throw exceptions, so this must be done in a `try...catch` statement.)

```

while(true) {
    int data = source.read();
    if (data < 0)
        break;
    copy.write(data);
}

```

The file-copy command in an operating system such as DOS or 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. For example, if the program is named `CopyFile` and if the user runs the program with the command `"java CopyFile work.dat oldwork.dat"`, then, in the program, `args[0]` will be the string `"work.dat"` and `args[1]` will be the string `"oldwork.dat"`. The value of `args.length` tells the program how many command-line arguments were specified by the user.

My `CopyFile` program gets the names of the files from the command-line arguments. It prints an error message and exits if the file names are not specified. To add a little interest, there are two ways to use the program. The command line can simply specify the two file names. In that case, if the output file already exists, the program will print an error message and end. This is to make sure that the user won't accidentally

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.*;

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(e.toString());
    }

} // end main()

} // end class CopyFile

```

Both of the previous programs use a command-line interface, but graphical user interface programs can also manipulate files. Programs typically have an "Open" command that reads the data from a file and displays it in a window and a "Save" command that writes the data from the window into a file. We can illustrate this in Java with a simple text editor program. The window for this program uses a `JTextArea` component to display some text that the user can edit. It also has a menu bar, with a "File" menu that includes "Open" and "Save" commands. To fully understand the examples in the rest of this section, you must be familiar with the material on menus and frames from [Section 7.7](#) and [Section 7.7](#). The examples also use file dialogs, which were introduced in [Section 2](#).

When the user selects the Save command from the File menu in the `TrivialEdit` program, the program pops up a file dialog box where the user specifies the file. The text from the `JTextArea` is written to the file. All this is done in the following instance method (where the variable, `text`, refers to the `TextArea`):

```
private void doSave() {
    // Carry out the Save command by letting the user specify
    // an output file and writing the text from the TextArea
    // to that file.
    File file; // The file that the user wants to save.
    JFileChooser fd; // File dialog that lets the user specify the file.
    fd = new JFileChooser(".");
    fd.setDialogTitle("Save Text As...");
    int action = fd.showSaveDialog(this);
    if (action != JFileChooser.APPROVE_OPTION) {
        // User has canceled, or an error occurred.
        return;
    }
    file = fd.getSelectedFile();
    if (file.exists()) {
        // If file already exists, ask before replacing it.
        action = JOptionPane.showConfirmDialog(this,
                                                "Replace existing file?");
        if (action != JOptionPane.YES_OPTION)
            return;
    }
    try {
        // Create a PrintWriter for writing to the specified
        // file and write the text from the window to that stream.
        PrintWriter out = new PrintWriter(new FileWriter(file));
        String contents = text.getText();
        out.print(contents);
        if (out.checkError())
            throw new IOException("Error while writing to file.");
        out.close();
    }
    catch (IOException e) {
        // Some error has occurred while trying to write.
        // Show an error message.
        JOptionPane.showMessageDialog(this,
                                      "Sorry, an error has occurred:\n" + e.getMessage());
    }
}
```

The methods `JOptionPane.showConfirmDialog()` and `JOptionPane.showMessageDialog()` were discussed in [Section 7.5](#).

When the user selects the Open command, a file dialog box allows the user to specify the file that is to be opened. It is assumed that the file is a text file. Since JTextAreas are not meant for displaying large amounts of text, the number of lines read from the file is limited to one hundred at most. Before the file is read, any text currently in the JTextArea is removed. Then lines are read from the file and appended to the JTextArea one by one, with a line feed character at the end of each line. This process continues until one hundred lines have been read or until the end of the input file is reached. If any error occurs during this process, an error message is displayed to the user in a dialog box. Here is the complete method:

```
private void doOpen() {
    // Carry out the Open command by letting the user specify
    // the file to be opened and reading up to 100 lines from
    // that file. The text from the file replaces the text
    // in the JTextArea.
    File file; // The file that the user wants to open.
    JFileChooser fd; // File dialog that lets the user specify a file.
    fd = new JFileChooser(new File("."));
    fd.setDialogTitle("Open File...");
    int action = fd.showOpenDialog(this);
    if (action != JFileChooser.APPROVE_OPTION) {
        // User canceled the dialog, or an error occurred.
        return;
    }
    file = fd.getSelectedFile();
    try {
        // Read lines from the file until end-of-file is detected,
        // or until 100 lines have been read. The lines are added
        // to the JTextArea, with a line feed after each line.
        TextReader in = new TextReader(new FileReader(file));
        String line;
        text.setText("");
        int lineCt = 0;
        while (lineCt < 100 && in.peek() != '\0') {
            line = in.getln();
            text.append(line + '\n');
            lineCt++;
        }
        if (in.eof() == false)
            text.append("\n\n***** Text truncated to 100 lines! *****\n");
        in.close();
    }
    catch (Exception e) {
        // Some error has occurred while trying to read the file.
        // Show an error message.
        JOptionPane.showMessageDialog(this,
            "Sorry, some error occurred:\n" + e.getMessage());
    }
}
```

The doSave () and doOpen () methods are the only part of the text editor program that deal with files. If you would like to see the entire program, you will find the source code in the file [TrivialEdit.java](#).

For a final example of files used in a complete program, you might want to look at [ShapeDrawWithFiles.java](#). This file defines one last version of the ShapeDraw program, which you last saw

in [Section 7.7](#). This version has a "File" menu for saving and loading the patterns of shapes that are created with the program. The program also serves as an example of using `ObjectInputStream` and `ObjectOutputStream`, which were discussed at the end of [Section 1](#). If you check, you'll see that the `Shape` class in this version has been declared to be `Serializable` so that objects of type `Shape` can be written to and read from object streams.

[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Section 10.4

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 still not quite as easy to work with as files are. But in Java, you can do network communication using input streams and output streams, just as you can use such streams to communicate with the user or to work with files. Nevertheless, opening a network connection between two computers is a bit tricky, since there are two computers involved and they have to somehow agree to open a connection. And when each computer can send data to the other, synchronizing communication can be a problem. But the fundamentals are the same as for other forms of I/O.

One of the standard Java packages is called `java.net`. This package includes several classes that can be used for networking. Two different styles of network I/O are supported. One of these, which is fairly high-level, is based on the World-Wide Web, and provides the sort of network communication capability that is used by a Web browser when it downloads pages for you to view. The main classes for this style of networking are `java.net.URL` and `java.net.URLConnection`. An object of type `URL` is an abstract representation of a **Universal Resource Locator**, which is an address for an HTML document or other resource on the Web. A `URLConnection` represents a network connection to such a resource.

The second style of I/O views the network at a lower level. It is based on the idea of a **socket**. A socket is used by a program to establish a connection with another program on a network. Communication over a network involves two sockets, one on each of the computers involved in the communication. Java uses a class called `java.net.Socket` to represent sockets that are used for network communication. The term "socket" presumably comes from an image of physically plugging a wire into a computer to establish a connection to a network, but it is important to understand that a socket, as the term is used here, is simply an object belonging to the class `Socket`. In particular, a program can have several sockets at the same time, each connecting it to another program running on some other computer on the network. All these connections use the same physical network connection.

This section gives a brief introduction to these basic networking classes, and shows how they relate to input and output streams and to exceptions.

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." See [Section 6.2](#) for more information.

An object belonging to the `URL` class represents such an address. Once you have a `URL` object, you can use it to open a `URLConnection` to the resource at that address. A url is ordinarily specified as a string, such as `"http://math.hws.edu/eck/index.html"`. There are also **relative url's**. A relative url specifies the location of a resource relative to the location of another url, which is called the **base** or **context** for the relative url. For example, if the context is given by the url `http://math.hws.edu/eck/`, then the incomplete, relative url `"index.html"` would really refer to `http://math.hws.edu/eck/index.html`.

An object of the class `URL` is not simply a string, but it can be constructed from a string representation of a url. A `URL` object can also be constructed from another `URL` object, representing a context, and a string that specifies a url relative to that context. These constructors have prototypes

```
public URL(String urlName) throws MalformedURLException
```

and

```
public URL(URL context, String relativeName)
    throws MalformedURLException
```

Note that these constructors will throw an exception of type `MalformedURLException` if the specified strings don't represent legal url's. The `MalformedURLException` class is a subclass of `IOException`, and it requires mandatory exception handling. That is, you must call the constructor inside a `try...catch` statement that handles the exception or in a subroutine that is declared to throw the exception.

The second constructor is especially convenient when writing applets. In an applet, two methods are available that provide useful URL contexts. The method `getDocumentBase()`, defined in the `Applet` class, returns an object of type `URL`. This URL represents the location from which the HTML page that contains the applet was downloaded. This allows the applet to go back and retrieve other files that are stored in the same location as that document. For example,

```
URL url = new URL(getDocumentBase(), "data.txt");
```

constructs a URL that refers to a file named `data.txt` on the same computer and in the same directory as the web page on which the applet is running. Another method, `getCodeBase()`, returns a URL that gives the location of the applet class file (which is not necessarily the same as the location of the document).

Once you have a valid URL object, you can call its `openConnection()` method to set up a connection. This method returns a `URLConnection`. The `URLConnection` object can, in turn, be used to create an `InputStream` for reading data from the resource represented by the URL. This is done by calling its `getInputStream()` method. For example:

```
URL url = new URL(urlAddressString);
URLConnection connection = url.openConnection();
InputStream in = connection.getInputStream();
```

The `openConnection()` and `getInputStream()` methods can both throw exceptions of type `IOException`. Once the `InputStream` has been created, you can read from it in the usual way, including wrapping it in another input stream type, such as `TextReader`. Reading from the stream can, of course, generate exceptions.

One of the other useful instance methods in the `URLConnection` class is `getContentType()`, which returns a `String` that describes the type of information available from the URL. The return value can be `null` if the type of information is not yet known or if it is not possible to determine the type. The type might not be available until after the input stream has been created, so you should generally call `getContentType()` after `getInputStream()`. The string returned by `getContentType()` is in a format called a **mime type**. Mime types include "text/plain", "text/html", "image/jpeg", "image/gif", and many others. All mime types contain two parts: a general type, such as "text" or "image", and a more specific type within that general category, such as "html" or "gif". If you are only interested in text data, for example, you can check whether the string returned by `getContentType()` starts with "text". (Mime types were first introduced to describe the content of email messages. The name stands for "Multipurpose Internet Mail Extensions." They are now used almost universally to specify the type of information in a file or other resource.)

Let's look at a short example that uses all this to read the data from a URL. This subroutine opens a connection to a specified URL, checks that the type of data at the URL is text, and the 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 Exception" and leaving it up to the main program to decide what to do when an error occurs.

```
static void readTextFromURL( String urlString ) throws Exception {
```

```

        // This subroutine attempts to copy text from the
        // specified URL onto the screen. All errors must
        // be handled by the caller of this subroutine.

    /* Open a connection to the URL, and get an input stream
       for reading data from the URL. */

    URL url = new URL(urlString);
    URLConnection connection = url.openConnection();
    InputStream urlData = connection.getInputStream();

    /* Check that the content is some type of text. */

    String contentType = connection.getContentType();
    if (contentType == null || contentType.startsWith("text") == false)
        throw new Exception("URL does not refer to a text file.");

    /* Copy characters from the input stream to the screen, until
       end-of-file is encountered (or an error occurs). */

    while (true) {
        int data = urlData.read();
        if (data < 0)
            break;
        System.out.print((char)data);
    }

} // end readTextFromURL()

```

A complete program that uses this subroutine can be found in the file [ReadURL.java](#). Here is an applet that does much the same thing (although it is more complex and uses some techniques that won't be covered until the [next section](#)). The applet lets you enter a URL. It can be a relative URL, which will be interpreted relative to the document base of the applet. Error messages or text loaded from the URL will be displayed in the text area of the applet. (The amount of text is limited to 10000 characters.) When the applet starts up, it is configured to load the file `ReadURL.java`. Just click the "Load" button:

(Applet "ReadURLApplet" would be displayed here
if Java were available.)

You can also try to use this applet to look at the HTML source code for this very page. Just type `s4.html` into the input box at the bottom of the applet and then click on the Load button. You might want to experiment with other urls to see what types of errors can occur. For example, entering `"bogus.html"` is likely to generate a `FileNotFoundException`, since no document of that name exists in the directory that contains this page. As another example, you can probably generate a `SecurityException` by trying to connect to `http://www.whitehouse.gov`. (Not because it's an official secret -- any url that does not lead back to the same computer from which the applet was loaded will generate a security exception. To protect you from malicious applets, an applet is allowed to open network connections only back to the computer from which it came.) The source code for the applet is in the file [ReadURLApplet.java](#)

Sockets, Clients, and Servers

Communication over the Internet is based on a pair of protocols called the **Internet Protocol** and the **Transmission Control Protocol**, which are collectively referred to as TCP/IP. (In fact, there is a more basic communication protocol called UDP that can be used instead of TCP in certain applications. UDP is supported in Java, but for this discussion, I'll stick to the full TCP/IP, which provides reliable two-way communication between networked computers.)

For two programs to communicate using TCP/IP, each program must create a socket, as discussed earlier in this section, and those sockets must be connected. Once such a connection is made, communication takes place using input streams and output streams. Each program has its own input stream and its own output stream. Data written by one program to its output stream is transmitted to the other computer. There, it enters the input stream of the program at the other end of the network connection. When that program reads data from its input stream, it is receiving the data that was transmitted to it over the network.

The hard part, then, is making a network connection in the first place. Two sockets are involved. To get things started, one program must create a socket that will wait passively until a connection request comes in from another socket. The waiting socket is said to be **listening** for a connection. On the other side of the connection-to-be, another program creates a socket that sends out a connection request to the listening socket. When the listening socket receives the connection request, it responds, and the connection is established. Once that is done, each program can obtain an input stream and an output stream for the connection. Communication takes place through these streams until one program or the other **closes** the connection.

A program that creates a listening socket is sometimes said to be a **server**, and the socket is called a **server socket**. A program that connects to a server is called a **client**, and the socket that it uses to make a connection is called a **client socket**. The idea is that the server is out there somewhere on the network, waiting for a connection request from some client. The server can be thought of as offering some kind of service, and the client gets access to that service by connecting to the server. This is called the **client/server model** of network communication. In many actual applications, a server program can provide connections to several clients at the same time. When a client connects to a server's listening socket, that socket does not stop listening. Instead, it continues listening for additional client connections at the same time that the first client is being serviced. (To do this, it is necessary to use "threads". We'll look at how it works in the [next section](#).)

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.

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`, then, 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.)

To use `Sockets` and `ServerSockets`, you need to know about Internet addresses. After all, a client program has to have some way to specify which computer, among all those on the network, it wants to communicate with. Every computer on the Internet has an **IP address** which identifies it uniquely among all the computers on the net. Many computers can also be referred to by **domain names** such as `math.hws.edu` or `www.whitehouse.gov`. (See [Section 1.7](#).) 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 socket is actually identified by a **port number** in combination with an IP address. A port number is just a 16-bit integer. A server does not simply listen for connections -- it listens for connections on a particular port. A potential client must know both the Internet address 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. If you create your own server programs, you should use port numbers greater than 1024.)

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
```

As soon as the `ServerSocket` is established, 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 thread that called the method can't do anything else. However, other threads in the same program can proceed.) The `ServerSocket` will continue listening for connections until it is closed, using its `close()` method, or until some error occurs.

Suppose that you want a server to listen on port 1728, and suppose that you've written a method `provideService(Socket)` to handle the communication with one client. Then the basic form of the server program would be:

```
try {
    ServerSocket server = new ServerSocket(1728);
    while (true) {
        Socket connection = server.accept();
        provideService(connection);
    }
} catch (IOException e) {
    System.out.println("Server shut down with error: " + e);
}
```

On the client side, a client socket is created using a constructor in the `Socket` class. To connect to a server on a known computer and port, you would use the constructor

```
public Socket(String computer, int port) throws IOException
```

The first parameter can be either an IP number or a domain name. This constructor will block until the connection is established or until an error occurs. Once the connection is established, you can use the `Socket` methods `getInputStream()` and `getOutputStream()` to obtain streams that can be used for communication over the connection. Keeping all this in mind, here is the outline of a method for working with a client connection:

```
void doClientConnection(String computerName, int listeningPort) {
    // ComputerName should give the name or ip number of the
    // computer where the server is running, such as
    // math.hws.edu. ListeningPort should be the port
    // on which the server listens for connections, such as 1728.
    Socket connection;
    InputStream in;
```

```

        OutputStream out;
        try {
            connection = new Socket(computerName,listeningPort);
            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 server.
        .
        try {
            connection.close();
            // (Alternatively, you might depend on the server
            // to close the connection.)
        }
        catch (IOException e) {
        }
    } // end doClientConnection()

```

All this makes network communication sound easier than it really is. (And if you think it sounded hard, then it's even harder.) If networks were completely reliable, things would be almost as easy as I've described. The problem, though, is to write robust programs that can deal with network and human error. I won't go into detail here -- partly because I don't really know enough about serious network programming in Java myself. 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. (Just don't try to write a replacement for Netscape.) Let's look at a few working examples of client/server programming.

Programming Examples

The first example consists of two programs. One is a simple client and the other is a matching server. The client makes a connection to the server, reads one line of text from the server, and displays that text on the screen. The text sent by the server consists of the current date and time on the computer where the server is running. In order to open a connection, the client must know the computer on which the server is running and the port on which it is listening. The server listens on port number 32007. The port number could be anything between 1025 and 65535, as long as the server and the client use the same port. Port numbers between 1 and 1024 are reserved for standard services and should not be used for other servers. The name or IP number of the computer on which the server is running must be specified as a command-line parameter. For example, if the server is running on a computer named math.hws.edu, then you would typically run the client with the command "java DateClient math.hws.edu". Here is the complete client program:

```

import java.net.*;
import java.io.*;

public class DateClient {

    static final int LISTENING_PORT = 32007;

    public static void main(String[] args) {

```



```

String computer;        // Name of the computer to connect to.
Socket connection;      // A socket for communicating with
                        // that computer.
Reader incoming;        // Stream for reading data from
                        // the connection.

/* Get computer name from command line. */

if (args.length > 0)
    computer = args[0];
else {
    // No computer name was given. Print a message and exit.
    System.out.println("Usage:  java DateClient <server>");
    return;
}

/* Make the connection, then read and display a line of text. */

try {
    connection = new Socket( computer, LISTENING_PORT );
    incoming = new InputStreamReader( connection.getInputStream() );
    while (true) {
        int ch = incoming.read();
        if (ch == -1 || ch == '\n' || ch == '\r')
            break;
        System.out.print( (char)ch );
    }
    System.out.println();
    incoming.close();
}
catch (IOException e) {
    TextIO.putln("Error:  " + e);
}

// end main()

} // end class DateClient

```

Note that all the communication with the server is done in a `try...catch`. This will catch the `IOExceptions` that can be generated when the connection is opened or closed and when characters are read from the stream. The stream that is used for input is a basic `Reader`, which includes the input operation `incoming.read()`. This function reads one character from the stream and returns its Unicode code number. If the end-of-stream has been reached, then the value `-1` is returned instead. The while loop reads characters and copies them to standard output until an end-of-stream or end-of-line is seen. An end of line is marked by one of the characters `\n` or `\r`, depending on the type of computer on which the server is running.

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 IP number `127.0.0.1` as referring to "this computer". That is, the command `java DateClient 127.0.0.1` will tell the `DateClient` program to connect to a server running on the same computer. Most computers will also recognize the name `localhost` as a name for "this computer".

The server program that corresponds to the `DateClient` client program is called `DateServe`. The `DateServe` 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 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. The subroutine creates a `PrintWriter` stream 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;

public class DateServe {

    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);
            TextIO.putln("Listening on port " + LISTENING_PORT);
            while (true) {
                connection = listener.accept();
                sendDate(connection);
            }
        }
        catch (Exception e) {
            TextIO.putln("Sorry, the server has shut down.");
            TextIO.putln("Error:  " + e);
            return;
        }

    } // end main()

    static void sendDate(Socket client) {
        // The parameter, client, is a socket that is
        // already connected to another program. Get
        // an output stream for the connection, send the
        // current date, and close the connection.
    }
}
```

```

    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 sendData()

} //end class DateServe

```

If you run `DateServe` in a command-line interface, it will sit and wait for connection requests and report them as they are received. To make the `DateServe` service permanently available on a computer, the program really should be run as a **daemon**. A daemon is a program that runs continually on a computer, independently of any user. The computer can be configured to start the daemon automatically as soon as the computer boots up. It then runs in the background, even while the computer is being used for other purposes. For example, a computer that makes pages available on the World Wide Web runs a daemon that listens for requests for pages and responds by transmitting the pages. It's just a souped-up analog of the `DateServe` 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 `out.println()` to send a line of data to the client, the server program calls `out.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 call this function every time you use an output stream to send data over a network connection. If you don't do so, it's possible that the stream will collect data until it has a large batch of data to send. This is done for efficiency, but it can impose unacceptable delays when the client is waiting for the transmission. It is even possible that some of the data might remain untransmitted when the socket is closed, so it is especially important to call `flush()` before closing the connection. This is one of those unfortunate cases where different implementations of Java can behave differently. If you fail to flush your output streams, it is possible that your network application will work on some types of computers but not on others.

In the `DateServe` example, the server transmits information and the client reads it. It's also possible to have two-way communication between client and server. As a first example, we'll look at a client and server that allow a user on each end of the connection to send messages to the other user. The program works in a command-line interface where the users type in their messages. In this example, the server waits for a connection from a single client and then closes down its listener so that no other clients can connect. After the client and server are connected, both ends of the connection work in much the same way. The user on the client end types a message, and it is transmitted to the server, which displays it to the user on that end. Then the user of the server types a message that is transmitted to the client. Then the client user types another message, and so on. This continues until one user or the other enters "quit" when prompted for a message. When that happens, the connection is closed and both programs terminate. The client program and the server program are very similar. The techniques for opening the connections differ, and the client is programmed to send the first message while the server is programmed to receive the first message. Here is the server program. You can find the client program in the file [CLChatClient.java](#). (The name "CLChat"

stands for command-line chat.)

```
import java.net.*;
import java.io.*;

public class CLChatServer {

    static final int DEFAULT_PORT = 1728;    // Port to listen on,
                                              // if none is specified
                                              // on the command line.

    static final String HANDSHAKE = "CLChat"; // 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 char MESSAGE = '0';        // This character is added to
                                              // the beginning of each message
                                              // that is sent.

    static final char CLOSE = '1';          // This character is sent to
                                              // the connected program when
                                              // the user quits.

    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.

        TextReader 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.

        /* 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) {
                TextIO.putln("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);
    TextIO.putln("Listening on port " + listener.getLocalPort());
    connection = listener.accept();
    listener.close();
    incoming = new TextReader(connection.getInputStream());
    outgoing = new PrintWriter(connection.getOutputStream());
    outgoing.println(HANDSHAKE);
    outgoing.flush();
    messageIn = incoming.getln();
    if (! messageIn.equals(HANDSHAKE) ) {
        throw new IOException("Connected program is not CLChat!");
    }
    TextIO.putln("Connected.  Waiting for the first message.\n");
}
catch (Exception e) {
    TextIO.putln("An error occurred while opening connection.");
    TextIO.putln(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 {
    while (true) {
        TextIO.putln("WAITING...");
        messageIn = incoming.getln();
        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) {
                TextIO.putln("Connection closed at other end.");
                connection.close();
                break;
            }
            messageIn = messageIn.substring(1);
        }
        TextIO.putln("RECEIVED:  " + messageIn);
        TextIO.put("SEND:      ");
        messageOut = TextIO.getln();
        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();
            TextIO.putln("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) {
    TextIO.putln("Sorry, an error has occurred.  Connection lost.");
    TextIO.putln(e.toString());
    System.exit(1);
}

} // end main()

} //end class CLChatServer

```

This program is a little more robust than DateServe. 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 client and server as part of setting up the connection. 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. When you design a client/server application, the design of the protocol is an important consideration. Another aspect of the CLChat protocol is that every line of text that is sent over the connection after the handshake 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 127.0.0.1" to connect to the server that is running on the same machine.

[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Section 10.5

Threads and Network Programming

NETWORK PROGRAMS ARE a natural application for threads. Threads were discussed in [Section 7.6](#) in the context of GUI programming. (If you have not already read that section, it would be a good idea to do it now.) As we saw in that section, a thread could be used in a GUI program to perform a long computation in parallel with the event-handling thread of the GUI. Network programs with graphical user interfaces can use the same technique: If a separate thread is used for network communication, then the communication can proceed in parallel with other things that are going on in the program. Threads are even more important in server programs. In many cases, a client can remain connected to a server for an indefinite period of time. It's not a good idea to make other potential clients wait during this period. A **multi-threaded server** starts a new thread for each client. Several threads can run at the same time, so several clients can be served at the same time. The second client doesn't have to wait until the server is finished with the first client. It's like a post office that opens up a new window for each customer, instead of making them all wait in line at one window.

Now, there are at least two problems with the command-line chat examples, `CLChatClient` and `CLChatServer`, from the [previous section](#). For one thing, after a user enters a message, the user must wait for a reply from the other side of the connection. It would be nice if the user could just keep typing lines and see the other user's messages 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 second problem has to do with opening connections in the first place. I can only run `CLChatClient` if I know that there is a `CLChatServer` running on some particular computer. Except in rather contrived situations, there is no way for me to know that. It would be nice if I could find out, somehow, who's out there waiting for a connection. In this section, we'll address both of these problems and, at the same time, learn a little more about network programming and about threads.

To address the first problem with the command-line chat programs, let's consider a GUI chat program. When one user connects to another user, a window should open on the screen with an input box where the user can enter messages to be transmitted to user on the other end of the connection. The user should be able to send a message at any time. The program should also be prepared to receive messages from the other side at any time, and those messages have to be displayed to the user as they arrive. In case this is not clear to you, here is an applet that simulates such a program. Enter a message in the input box at the bottom of the applet, and press return (or, equivalently, click the "Send" button):

(Applet "ChatSimulation" would be displayed here
if Java were available.)

Both incoming messages and messages that you send are posted to the `JTextArea` that occupies most of the applet. This is not a real network connection. When you send your first message, a separate thread is started by the applet. This thread *simulates* incoming messages from the other side of a network connection. In fact, it just chooses the messages at random from a pre-set list. At the same time, you can continue to enter and send messages. The `run()` method that is executed by the thread carries out the following algorithm:

```
Post the message "Hey, hello there! Nice to chat with you."
while(running):
    Wait a random time, between 2 and 12 seconds
    Select a random message from the list
    Post the selected message in the JTextArea
```

The variable `running` is set to `false` when the applet is stopped, as a signal to the thread that it should

exit. The thread is created and started in the `actionPerformed` method that responds when you press return or click the "Send" button for the first time. You can find the complete source code in the file [ChatSimulation.java](#), but I really want to look at the programming for the real thing rather than the simulation. The GUI chat program that we will look at is [ChatWindow.java](#). The interface in this program will look similar to the simulation, but there will be a real network connection, and the incoming messages will be coming from the other side of that connection. The basic idea is not much more complicated than the simulation. A separate thread is created to wait for incoming messages and post them as they arrive. The `run()` method for this thread has an outline that is similar to the one for the simulation:

```
while the connection is open:
    Wait for a message to arrive from the other side
    Post the message in the JTextArea
```

However, the whole thing is complicated by the problem of opening and closing the connection and by the input/output errors that can occur at any time. The `ChatWindow` class is fairly sophisticated, and I don't want to cover everything that it does, but I will describe some of its functions. You should read the source code if you want to understand it completely.

First, there is the question of how a connection can be established between two `ChatWindows`. As the `ChatWindow` class is designed, the connection must be established before the window is opened. Recall that one end of a network connection is represented by an object of type `Socket`. The connected `Socket` is passed as a parameter to the `ChatWindow` constructor. This makes `ChatWindow` into a nicely reusable class that can be used in a variety of programs that set up the connection in different ways. The simplest approach to establishing the connection uses a command-line interface, just as is done with the `CLChat` programs. Once the connection has been established, a `ChatWindow` is opened on each side of the connection, and the actual chatting takes place through the windows instead of the command line. For this example, I've written a `main()` routine that can act as either the server or the client, depending on the command line argument that it is given. If the first command line argument is "-s", the program will act as a server. Otherwise, it assumes that the first argument specifies the computer where the server is running, and it acts as a client. The code for doing this is:

```
try {
    if (args[0].equalsIgnoreCase("-s")) {
        // Act as a server.  Wait for a connection.
        ServerSocket listener = new ServerSocket(port);
        System.out.println("Listening on port "
                           + listener.getLocalPort());
        connection = listener.accept();
        listener.close();
    }
    else {
        // Act as a client.  Request a connection with
        // a server running on the computer specified in args[0].
        connection = new Socket(args[0],port);
    }
    out = new PrintWriter(connection.getOutputStream());
    out.println(HANDSHAKE);
    out.flush();
    in = new TextReader(connection.getInputStream());
    message = in.getln();
    if (! message.equals(HANDSHAKE) ) {
        throw new IOException(
            "Connected program is not a ChatWindow");
    }
    System.out.println("Connected.");
}
}
```

```

    catch (Exception e) {
        System.out.println("Error opening connection.");
        System.out.println(e.toString());
        return;
    }

    ChatWindow w;    // The window for this end of the connection.
    w = new ChatWindow("ChatWindow", connection);

```

As it happens, I've taken the rather twisty approach of putting this `main()` routine in the `ChatWindow` class itself. (Possibly, it would be better style to put the `main()` routine in a different class.) This means that you can run `ChatWindow` as a standalone program. If you run it with the command `"java ChatWindow -s"`, it will run as a server. To run it as a client, use the command `"java ChatWindow <server>"`, where `<server>` is the name or IP number of the computer where the server is running. Use `"localhost"` as the name of the server, if you want to test the program by connecting to a server running on the same computer as the client. Whether the program is running as a client or as a server, once a connection is made, the window will open, and you can start chatting.

The constructor for the `ChatWindow` has the job of starting a thread to handle incoming messages. It also creates input and output streams for sending and receiving. The part of the constructor that performs these tasks look like this (with just a few changes for the sake of simplicity):

```

    try {
        incoming = new TextReader( connection.getInputStream() );
        outgoing = new PrintWriter( connection.getOutputStream() );
        // Here, connection is the Socket that will be used for
        // communication.  Input and output streams are created
        // for writing and reading information over the connection.
    }
    catch (IOException e) {
        // An error occurred while trying to get the streams.
        // Set up user interface to reflect the error.  The
        // "transcript" is the JTextArea where messages are displayed.
        transcript.setText("Error opening I/O streams!\n"
            + "Connection can't be used.\n"
            + "You can close the window now.\n");
        sendButton.setEnabled(false);
        connection = null;
    }

    /* Create the thread for reading data from the connection,
       unless an error just occurred. */

    if (connection != null) {
        // Create a thread to execute the run() method in this
        // applet class, and start the thread.  The run() method
        // will wait for incoming messages and post them to the
        // transcript when they are received.
        reader = new Thread(this);
        reader.start();
    }

```

The input stream, `incoming`, is used by the thread to read messages from the other side of the connection. It does this simply by saying `incoming.getln()`. This command will not return until a line of text has been received or until an error occurs. The output stream, `outgoing`, is used by the `actionPerformed()` method to transmit the text from the text input box.

When either user closes his `ChatWindow`, the connection must be closed on *both* sides. The connection might also be closed because an error occurs, such as a network failure. It takes some care to handle all this correctly. Take a look at the [source code](#) if you are interested.

There is still a big problem with running `ChatWindow` in the way I've just described. Suppose I want to set up a connection. How do I know who has a `ChatWindow` running as a server? If I start up the server myself, how will anyone know about it? The `CLChat` programs have the same problem. What I would like is a program that would show me a list of all the "chatters" who are available, and I would like to be able to add myself to the list so that other people can tell that I am available to receive connections. The problem is, who is going to keep the list and how will my program get a copy of the list?

This is a natural application for a server! We can have a server whose job is to keep a list of available chatters. This server can be run as a daemon on a "well-known computer", so that it is always available at a known location on the Internet. Then, a program anywhere on the Internet can contact the server to get the list of chatters or to register a person on the list. That program acts as a client for the server.

In fact, I've written such a server program. It's called `ConnectionBroker`, and the source code is available in the file [ConnectionBroker.java](#). The `main()` routine of this server is similar to the `main()` routine of the `DateServe` example that was given at the beginning of this section. That is, it runs in an infinite loop in which it accepts connections and processes them. In this case, however, the processing of each request is much more complicated and can take a long time, so the `main()` routine sets up a separate thread to process each connection request. That's all the main routine does with the connection. The thread takes care of all the details, while the main program goes on to the next connection request. Here is the `main()` routine from `ConnectionBroker`:

```
public static void main(String[] args) {
    // The main() routine creates a listening socket and
    // listens for requests.  When a request is received,
    // a thread is created to service it.
    int port; // Port on which server listens.
    ServerSocket listener;
    Socket client;
    if (args.length == 0)
        port = DEFAULT_PORT;
    else {
        try {
            port = Integer.parseInt(args[0]);
        }
        catch (NumberFormatException e) {
            System.out.println(args[0] + " is not a legal port number.");
            return;
        }
    }
    try {
        listener = new ServerSocket(port);
    }
    catch (IOException e) {
        System.out.println("Can't start server.");
        System.out.println(e.toString());
        return;
    }
    System.out.println("Listening on port " + listener.getLocalPort());
    try {
```

```

        while (true) {
            client = listener.accept(); // Get a connection request.
            new ClientThread(client); // Start a thread to handle it.
        }
    }
    catch (Exception e) {
        System.out.println("Server shut down unexpectedly.");
        System.out.println(e.toString());
        System.exit(1);
    }
}

```

Once the processing thread has been started to handle the connection, the thread reads a command from the client, and carries out that command. It understands three types of commands:

- A REGISTER command that adds the client to the list of available chatters. The server keeps this list in an internal data structure. The connection remains open and the thread waits for some other user to request a connection with that client. Once a connection is made, the client is removed from the list.
- A SEND_CLIENT_LIST command requests a copy of the list of available chatters. The server responds by sending the list and closing the connection.
- A CONNECT command requests the server to set up a connection with one of the chatters in the list. The server sets up the connection, and -- if no error occurs -- informs both parties that a connection has been established. The two parties can then start sending messages to each other. (These messages actually continue to pass through the server. The direct network connections are between the server and the two clients. The server relays messages from each client to the other. It's done this way so that a ConnectionBroker will work with applets as clients, as long as the applets are loaded from the computer where the server is running. An applet is not ordinarily allowed to make network connections, except to the computer from which it was loaded.)

To use a ConnectionBroker, you need a program that acts as a client for the ConnectionBroker service. I have an applet that does this. The applet tries to connect to a ConnectionBroker server on the computer from which the applet was loaded. If no such server is running on that computer, the applet will display an error notification saying that it can't connect to the server. You are likely to get an error message unless you have downloaded this on-line textbook and are reading the copy on your own computer. In that case, you should be able to run the ConnectionBroker server on your computer and use the applet to connect to it. (Just compile ConnectionBroker.java and then give the command "java ConnectionBroker" in the same directory. It will print out "Listening on port 3030" and start waiting for connections. You will have to abort the program in some way to get it to end, such as by hitting CONTROL-C.) Here is the applet:

(Applet "BrokeredChat" would be displayed here
if Java were available.)

If the applet does find a server, it will display the list of available chatters in the JComboBox on the third line of the applet. If no chatters are available on the server, then you'll just see the message "(None available)". Once you register yourself, you will be included in this list, and you can open a connection to yourself. (Not a very interesting conversation perhaps, but it will demonstrate how the program works.) The procedures for registering yourself with the server and for requesting a connection to someone in the JComboBox should be easy enough to figure out. When you register yourself, a ChatWindow will open and will wait for someone to connect to you. A ChatWindow will also open when you request a connection.

You can enter yourself multiple times in the list, if you want, and you can connect to multiple people on the list. A separate ChatWindow will open for each connection.

This networked chat application is still very much a demonstration system. That is, it is not robust enough

or full-featured enough to be used for serious applications. I tried to keep the interactions among the server, the applet, and the connection windows simple enough to understand with a reasonable effort. If you are interested in pursuing the topic of network programming, I suggest you start by reading the three source code files for this example: the applet [BrokeredChat.java](#), the server [ConnectionBroker.java](#), and the window [ChatWindow.java](#).

The Problem of Synchronization

Although I don't want to say too much about the `ConnectionBroker` program, there is still one general question I want to look at: What happens when two or more threads use the same data? When this is the case, it's possible for the data to become corrupted, unless access to the data is carefully **synchronized**. The problem arises when two threads both try to access the data at the same time, or when one thread is interrupted by another when it is in the middle of accessing the data. Synchronization is used to make sure that this doesn't happen. To see what can go wrong, let's look at a typical example: a bank account. Suppose that the amount of money in a bank account is represented by the class:

```
public class BankAccount {
    private double balance; // amount of money in account
    public double getBalance() {
        return balance;
    }
    public void withdraw(double amount) {
        // Precondition: The balance is >= the amount.
        balance = balance - amount;
    }
    .
    . // Other methods
    .
}
```

Suppose that `account` is an object of type `BankAccount`, and that this variable is used by several threads. Suppose that one of these threads wants to do a withdrawal of \$100. This should be easy:

```
if ( account.getBalance() >= 100)
    account.withdraw();
```

But suppose that two threads try to do a withdrawal at the same time from an account that contains \$150. It might happen that one thread calls `account.getBalance()` and gets a balance of 150. But at that moment, the first thread is interrupted by the other thread. The other thread calls `account.getBalance()` and also gets 150 as the balance. Both threads decide it's safe to withdraw \$100, but when they do so, the balance drops below zero. Actually, it's even worse than this. The statement "`balance = balance - amount`" is actually executed as several steps: Read the balance; subtract the amount; store the new balance. It's possible for a thread to be interrupted in the middle of this. Suppose that two threads try to withdraw \$100. If they execute the withdrawal at about the same time, it might happen that the order of operations is:

1. First thread reads the balance, and gets \$150
2. Second thread reads the balance, and gets \$150
3. Second thread subtracts \$100 from \$150, leaving \$50
4. Second thread stores the new balance, \$50
5. First thread (continuing after interruption)
 - subtracts \$100 from \$150, leaving \$50
6. First thread stores the new balance, \$50

The net result is that even though there have been *two* withdrawals of \$100, the amount in the account has

only gone down by one hundred. The bank will probably not be very happy with its programmers!

You might not think that this sequence of events is very likely, but when large numbers of computations are being performed by several threads on shared data, problems like this are almost certain to occur, and they can be disastrous when they happen. The synchronization problem is very real: Access to shared data must be controlled.

As I mentioned in [Section 7.6](#), the Swing GUI library solves the synchronization problem in a straightforward way: Only one thread is allowed to change the data used by Swing components. That thread is the event-handling thread. If some other thread wants to do something with a Swing component, it's not allowed to do it itself. It must arrange for the event-handling thread to do it instead. Swing has methods `SwingUtilities.invokeLater()` and `SwingUtilities.invokeAndWait()` to make this possible. This is the only type of synchronization that is used in the `ChatSimulation`, `ChatWindow`, and `BrokeredChat` programs.

In many cases, Swing's solution to the synchronization problem is not applicable and might even defeat the purpose of using multiple threads in the first place. Java has a more general means for controlling access to shared data. It's done using a new type of statement: the **synchronized statement**. A synchronized statement has the form:

```
synchronized ( <object-reference> ) {
    <statements>
}
```

For example:

```
synchronized(account) {
    if ( account.getBalance() >= amount )
        balance = balance - amount;
}
```

The idea is that the **<object-reference>** -- `account` in the example -- is used to "lock" access to the statements. Each object in Java has a lock that can be used for synchronization. When a thread executes `synchronized(account)`, it takes possession of `account`'s lock, and will hold that lock until it is done executing the statements inside the `synchronized` statement. If a second thread tries to execute `synchronized(account)` while the first thread holds the lock, the second thread will have to wait until the first thread releases the lock. This means that it's impossible for two different threads to execute the statements in the `synchronized` statement at the same time. The scenarios that we looked at above, which could corrupt the data, are impossible.

It's possible to use the same object in two different `synchronized` statements. Only *one* of those statements can be executed at any given time, because all the statements require the same lock before they can be executed. By putting *every* access to some data inside `synchronized` statements, and using the same object for synchronization in each statement, we can make sure that that data will only be accessed by one thread at a time. This is the general approach for solving the synchronization problem. It is an approach that will work for multi-threaded servers, such as `ConnectionBroker`, where there are many threads that might need access to the same data. The `ConnectionBroker` program, for example, keeps a list of clients in a `Vector` named `clientList`. This vector is used by many threads, and access to it must be controlled. This is accomplished by putting all access to the vector in `synchronized` statements. The vector itself is used as the synchronization object (although there is no rule that says that the synchronization object has to be the same as the data that is being protected). Here, for your amusement is all the code from `ConnectionBroker.java` that accesses `clientList`:

```
/* These four methods synchronize access to a Vector, clientList,
   which contains a list of the clients of this server. The
   synchronization also protects the variable nextClientInfo. */
```



```

static void addClient(Client client) {
    // Adds a new client to the clientList vector.
    synchronized(clientList) {
        client.ID = nextClientID++;
        if (client.info.length() == 0)
            client.info = "Anonymous" + client.ID;
        clientList.addElement(client);
    }
    System.out.println("Added client " + client.ID + " " + client.info);
}

static void removeClient(Client client) {
    // Removes the client from the clientList, if present.
    synchronized(clientList) {
        clientList.removeElement(client);
    }
    System.out.println("Removed client " + client.ID);
}

static Client getClient(int ID) {
    // Removes client from the clientList vector, if it
    // contains a client of the given ID.  If so, the
    // removed client is returned.  Otherwise, null is returned.
    synchronized(clientList) {
        for (int i = 0; i < clientList.size(); i++) {
            Client c = (Client)clientList.elementAt(i);
            if (c.ID == ID) {
                clientList.removeElementAt(i);
                System.out.println("Removed client " + c.ID);
                c.ID = 0; // Since this client is no longer waiting!
                return c;
            }
        }
        return null;
    }
}

static Client[] getClients() {
    // Returns an array of all the clients in the
    // clientList.  If there are none, null is returned.
    synchronized(clientList) {
        if (clientList.size() == 0)
            return null;
        Client[] clients = new Client[ clientList.size() ];
        for (int i = 0; i < clientList.size(); i++)
            clients[i] = (Client)clientList.elementAt(i);
        return clients;
    }
}

```

You don't have to understand exactly what is going on here, just that the `synchronized` statements are used to control access to data that is being shared by multiple threads. There is much more to learn about threads; synchronization is only one of the problems that arise. However, I will leave the topic here. (One reason why I covered this much was to fulfill a promise made back in [Section 3.6](#), where there was a list of all the different types of statements in Java. The `synchronized` statement was the last of these that we needed to cover.)

End of Chapter 10

[[Next Chapter](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Programming Exercises For Chapter 10

THIS PAGE CONTAINS programming exercises based on material from [Chapter 10](#) of this [on-line Java textbook](#). Each exercise has a link to a discussion of one possible solution of that exercise.

Exercise 10.1: The [WordList](#) program from [Section 10.3](#) reads a text file and makes an alphabetical list of all the words in that file. The list of words is output to another file. Improve the program so that it also keeps track of the number of times that each word occurs in the file. Write two lists to the output file. The first list contains the words in alphabetical order. The number of times that the word occurred in the file should be listed along with the word. Then write a second list to the output file in which the words are sorted according to the number of times that they occurred in the files. The word that occurred most often should be listed first.

[See the solution!](#)

Exercise 10.2: 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.

[See the solution!](#)

Exercise 10.3: [Section 8.4](#) presented a `PhoneDirectory` class as an example. A `PhoneDirectory` holds a list of names and associated phone numbers. But a phone directory is pretty useless unless the data in the directory can be saved permanently -- that is, in a file. Write a phone directory program that keeps its list of names and phone numbers in a file. The user of the program should be able to look up a name in the directory to find the associated phone number. The user should also be able to make changes to the data in the directory. Every time the program starts up, it should read the data from the file. Before the program terminates, if the data has been changed while the program was running, the file should be re-written with the new data. Designing a user interface for the program is part of the exercise.

[See the solution!](#)

Exercise 10.4: 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 <file>", where <file> is a file name. The server checks whether the requested file actually exists. If so, it first sends the word "ok" as a message to the client. Then it sends the contents of the file and closes the connection. Otherwise, it sends the word "error" to the client and closes the connection.

Ideally, your server should start a separate thread to handle each connection request. However, if you don't want to deal with threads you can just call a subroutine to handle the request. See the `DirectoryList` example in [Section 10.2](#) for help with the problem of getting the list of files in the directory.

[See the solution!](#)

Exercise 10.5: Write a client program for the server from Exercise 10.4. Design a user interface that will let the user do at least two things: Get a list of files that are available on the server and display the list on standard output. 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!](#)

[[Chapter Index](#) | [Main Index](#)]

Quiz Questions For Chapter 10

THIS PAGE CONTAINS A SAMPLE quiz on material from [Chapter 10](#) of this [on-line Java textbook](#). You should be able to answer these questions after studying that chapter. Sample answers to all the quiz questions can be found [here](#).

Question 1: In Java, input/output is done using streams. Streams are an *abstraction*. Explain what this means and why it is important.

Question 2: Java has two types of streams: 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: Explain what is meant by the *client / server* model of network communication.

Question 7: What is a *Socket*?

Question 8: What is a *ServerSocket* and how is it used?

Question 9: Network server programs are often *multithreaded*. Explain what this means and why it is true.

Question 10: 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.

[[Answers](#) | [Chapter Index](#) | [Main Index](#)]

Chapter 11

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. That is, 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 11:

- Section 1: [Recursion](#)
 - Section 2: [Linking Objects](#)
 - Section 3: [Stacks and Queues](#)
 - Section 4: [Binary Trees](#)
 - Section 5: [A Simple Recursive-descent Parser](#)
 - [Programming Exercises](#)
 - [Quiz on this Chapter](#)
-

[[First Section](#) | [Next Chapter](#) | [Previous Chapter](#) | [Main Index](#)]

Section 11.1

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 be other sets. A "statement" in Java can be a `while` statement, which is made up of the word "while", a boolean-valued condition, and a statement.

Recursive definitions can describe very complex situations with just a few words. A definition of the term "ancestor" without using recursion might go something like "a parent, or a grandparent, or a great-grandparent, or a great-great-grandparent, and so on." But saying "and so on" is not very rigorous. (I've often thought that recursion is really just a rigorous way of saying "and so on.") You run into the same problem if you try to define a "directory" as "a file that is a list of files, where some of the files can be lists of files, where some of **those files can be lists of files, and so on.**" **Trying to describe what a Java statement can look like, without using recursion in the definition, would be difficult and probably pretty comical.**

Recursion can be used as a programming technique. A **recursive subroutine** is one that calls itself, either directly or indirectly. To say that a subroutine calls itself directly means that its definition contains a subroutine call statement that calls the subroutine that is being defined. To say that a subroutine calls itself indirectly means that it calls a second subroutine which in turn calls the first subroutine (either directly or indirectly). A recursive subroutine can define a complex task in just a few lines of code. In the rest of this section, we'll look at a variety of examples, and we'll see other examples in the remaining sections of this chapter.

Let's start with an example that you've seen before: the binary search algorithm from [Section 8.4](#). 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 [Section 9.2](#), 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: We can say immediately that the value does not occur in the list. 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:

```
static int binarySearch(int[] A, int loIndex, int hiIndex, int value) {
    // Search in the array A in positions from loIndex to hiIndex,
    // inclusive, for the specified value. It is assumed that the
    // array is sorted into increasing order. If the value is
    // found, return the index in the array where it occurs.
    // If the value is not found, return -1.

    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()
```

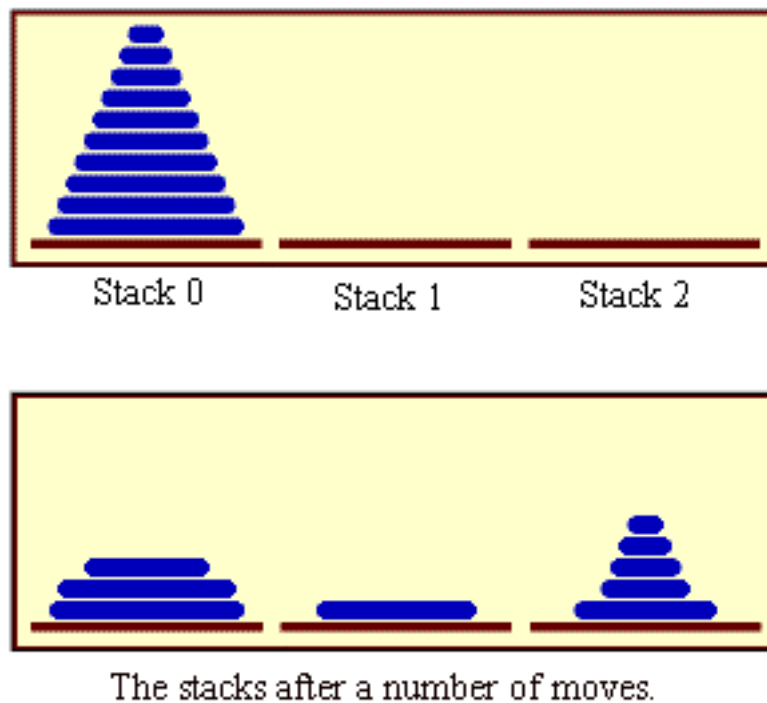
In this routine, the parameters `loIndex` and `hiIndex` specify the part of the array that is to be searched. To search an entire array, it is only necessary to call `binarySearch(A, 0, A.length - 1, value)`. In the two base cases -- where there are no elements in the specified range of indices and when 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`.)

Binary search can be implemented with a `while` loop, instead of with recursion, as was done in [Section 8.4](#). 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 situation for a stack of ten disks is shown in the top half of the following picture. The situation after a number of moves have been made is shown in the bottom half of the picture. These pictures are from the applet at the end of [Section 10.5](#), which displays an animation of the step-by-step solution of the problem.



The problem is to move ten disks from Stack 0 to Stack 1, subject to certain rules. Stack 2 can be used 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:

```
void TowersOfHanoi(int disks, int from, int to, int spare) {
    // 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.
    if (disks == 1) {
        // There is only one disk to be moved. Just move it.
        System.out.println("Move a disk from stack number "
            + from + " to stack number " + to);
    }
    else {
        // Move all but one disk to the spare stack, then
        // move the bottom disk, then put all the other
        // disks on top of it.
        TowersOfHanoi(disks-1, from, spare, to);
        System.out.println("Move a disk from stack number "
            + from + " to stack number " + to);
        TowersOfHanoi(disks-1, spare, to, from);
    }
}
```

This subroutine just expresses the natural recursive solution. The recursion works because each recursive call involves a smaller number of disks, and the problem is trivial to solve in the base case, when there is only one disk. To solve the "top level" problem of moving N disks from Stack 0 to Stack 1, it should be called with the command `TowersOfHanoi(N, 0, 1, 2)`. Here is an applet that uses this subroutine. You can specify the number of disks. Be careful. The number of steps increases rapidly with the number of disks.

(Applet "TowersOfHanoiConsole" would be displayed here
if Java were available.)

What this applet shows you is a mass of detail that you don't really want to think about! The difficulty of following the details contrasts sharply with the simplicity and elegance of the recursive solution. Of course, you really want to leave the details to the computer. It's much more interesting to watch the applet from Section 10.5, which shows the solution graphically. That applet uses the same recursive subroutine, except that the `System.out.println` statements are replaced by commands that show the image of the disk being moved from one stack to another. You can find the complete source code in the file [TowersOfHanoi.java](#).

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.

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 8.4](#), are fairly simple, but they are rather slow when applied to large arrays. Faster sorting algorithms are available. One of these is Quicksort, a recursive algorithm which turns out to be the fastest sorting algorithm in most

situations.

The Quicksort algorithm is based on a simple but clever idea: Given a list of items, select any item from the list. This item is called the **pivot**. (In practice, I'll just use the first item in the list.) Move all the items that are smaller than the pivot to the beginning of the list, and move all the items that are larger than the pivot to the end of the list. Now, put the pivot between the two groups of items. This puts the pivot in the position that it will occupy in the final, completely sorted array. It will not have to be moved again. We'll refer to this procedure as QuicksortStep.

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 12 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 number 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.

```
static int quicksortStep(int[] A, int lo, int hi) {
    // 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.

    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 it used
    // to hold the pivot.

    while (hi > lo) {

        while (hi > lo && A[hi] > pivot) {
            // Move hi down past numbers greater than pivot.
            // These numbers do not have to be moved.
            hi--;
        }
    }
}
```

```

        if (hi == lo)
            break;

        // The number A[hi] is less than pivot.  Move it into
        // the available space at A[lo], leaving an available
        // space at A[hi].

        A[lo] = A[hi];
        lo++;

        while (hi > lo && A[lo] < pivot) {
            // Move lo up past numbers less than pivot.
            // These numbers do not have to be moved.
            lo++;
        }

        if (hi == lo)
            break;

        // The number A[lo] is greater than pivot.  Move it into
        // the available space at A[hi], leaving an available
        // space at A[lo].

        A[hi] = A[lo];
        hi--;

    } // end while

    // At this point, lo has become equal to hi, and there is
    // an available space at that position.  This position lies
    // between numbers less than pivot and numbers greater than
    // pivot.  Put pivot in this space and return its location.

    A[lo] = pivot;
    return lo;

} // end QuicksortStep

```

With this subroutine in hand, Quicksort is easy. The Quicksort algorithm for sorting a list consists of applying QuicksortStep to the list, then applying Quicksort recursively to the items that lie to the left of the pivot and to the items that lie to the right of the pivot. 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.

```

static void quicksort(int[] A, int lo, int hi) {
    // Apply quicksort to put the array elements between
    // position lo and position hi into increasing order.
    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 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)`.

Here's an applet that shows a grid of little squares. The gray squares are "filled" and the white squares are "empty." For the purposes of this applet, 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 you click on any filled square in the applet, the computer will count the squares in the blob that contains it, and it will change the color of those squares to red. Click on the "New Blobs" button to create a new random pattern in the grid. The pop-up menu gives the approximate percentage of squares that will be filled in the new pattern. The more filled squares, the larger the blobs. The button labeled "Count the Blobs" will tell you how many different blobs there are in the pattern.

Sorry, but your browser
doesn't support Java.

Recursion is used in this applet to count the number of squares in a blob. Without recursion, this would be a very difficult thing to program. 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 applet uses a recursive instance method named `getBlobSize()` to count the number of squares in the blob that contains the square in a given row `r` and column `c`. If there is no filled square at position (r, c) , then the answer is zero. Otherwise, `getBlobSize()` has to count all the filled squares that can be reached from the square at position (r, c) . The idea is to use `getBlobSize()` recursively to get the number of filled squares that can be reached from each of the neighboring positions, $(r+1, c)$, $(r-1, c)$, $(r, c+1)$, and $(r, c-1)$. Add up these numbers, and add one to count the square at (r, c) itself, and you get the total number of filled squares that can be reached from (r, c) . Here is an implementation of this algorithm, as stated. Unfortunately, it has a serious flaw: It leads to an infinite recursion!

```

int getBlobSize(int r, int c) { // BUGGY, INCORRECT VERSION!!
    // This INCORRECT method tries to count all the filled
    // squares that can be reached from position (r,c) in the grid.
    if (r < 0 || r >= rows || c < 0 || c >= columns) {
        // This position is not in the grid, so there is
        // no blob at this position. Return a blob size of zero.
        return 0;
    }
    if (filled[r][c] == false) {
        // This square is not part of a blob, so return zero.
        return 0;
    }
}

```

```

    }
    int size = 1; // Count the square at this position, then count the
                  // the blobs that are connected to this square
                  // horizontally or vertically.
    size += getBlobSize(r-1,c);
    size += getBlobSize(r+1,c);
    size += getBlobSize(r,c-1);
    size += getBlobSize(r,c+1);
    return size;
} // end INCORRECT getBlobSize()

```

Unfortunately, this routine will count the same square more than once. In fact, it will try to count each square infinitely often! Think of yourself standing at position (r, c) and trying to follow these instructions. The first instruction tells you to move up one row. You do that, and then you apply the same procedure. As one of the steps in that procedure, you have to move down one row and apply the same procedure yet again. But that puts you back at position (r, c) ! From there, you move up one row, and from there you move down one row.... Back and forth forever! We have to make sure that a square is only counted and processed once, so we don't end up going around in circles. The solution is to leave a trail of breadcrumbs -- or on the computer a trail of boolean variables -- 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!

Another 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 is already visited, it doesn't count it or process it further. The technique of "marking" items as they are encountered is one that used over and over in the programming of recursive algorithms. Here is the corrected version of `getBlobSize()`, with changes shown in red:

```

int getBlobSize(int r, int c) {
    // 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.
    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 applet, this method is used to determine the size of a blob when the user clicks on a square. After `getBlobSize()` has performed its task, all the squares in the blob are still marked as visited. The `paintComponent()` method draws visited squares in red, which makes the blob visible. The `getBlobSize()` method is also used for counting blobs. This is done by the following method, which includes comments to explain how it works:

```

void countBlobs() {
    // 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.

    int count = 0; // Number of blobs.

    /* First clear out the visited array. The getBlobSize() method
       will mark every filled square that it finds by setting the
       corresponding element of the array to true. Once a square
       has been marked as visited, it will stay marked until all the
       blobs have been counted. This will prevent the same blob from
       being counted more than once. */

    for (int r = 0; r < rows; r++)
        for (int c = 0; c < columns; c++)
            visited[r][c] = false;

    /* For each position in the grid, call getBlobSize() to get the
       size of the blob at that position. If the size is not zero,
       count a blob. Note that if we come to a position that was part
       of a previously counted blob, getBlobSize() will return 0 and
       the blob will not be counted again. */

    for (int r = 0; r < rows; r++)
        for (int c = 0; c < columns; c++) {
            if (getBlobSize(r,c) > 0)
                count++;
        }

    repaint(); // Note that all the filled squares will be red,
               // since they have all now been visited.

    message.setText("The number of blobs is " + count);

} // end countBlobs()

```

You can find the complete source code for the applet in the file [Blobs.java](#).

Among the decorative end-of-chapter applets in this text, there are two others that use recursion: The maze-solving applet from the end of [Section 8.5](#) and the pentominos applet from the end of [Section 5](#) in this chapter.

The Maze applet first builds a random maze. It then tries to solve the maze by finding a path through the maze from the upper left corner to the lower right corner. This problem is actually very similar to the blob-counting problem. The recursive maze-solving routine starts from a given square, and it visits each neighboring square and calls itself recursively from there. The recursion ends if the routine finds itself at the lower right corner of the maze.

The Pentominos applet is an implementation of a classic puzzle. A pentomino is a connected figure made up of five equal-sized squares. There are exactly twelve figures that can be made in this way, not counting all the possible rotations and reflections of the basic figures. The problem is to place the twelve pentominos on an 8-by-8 board in which four of the squares have already been marked as filled. The recursive solution looks at a board that has already been partially filled with pentominos. The subroutine looks at each remaining piece in turn. It tries to place that piece in the next available place on the board. If the piece fits, it calls itself recursively to try to fill in the rest of the solution. If that fails, then the subroutine goes on to the next piece. (By the way, if you click on the Pentominos applet, it will start over with a new, randomly chosen problem.)

The Maze applet and the Pentominos applet are fun to watch, and they give nice visual representations of recursion. We'll encounter other examples of recursion later in this chapter.

[[Next Section](#) | [Previous Chapter](#) | [Chapter Index](#) | [Main Index](#)]

Section 11.2

Linking Objects

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.

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:

```
class Employee {
    // An object of type Employee holds data about
    //     one 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!" );
}
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
            runner = runner.supervisor;
        }
        System.out.println( "There are " + count
                            + " supervisors between " + emp.name
                            + " and the boss." );
    }
}

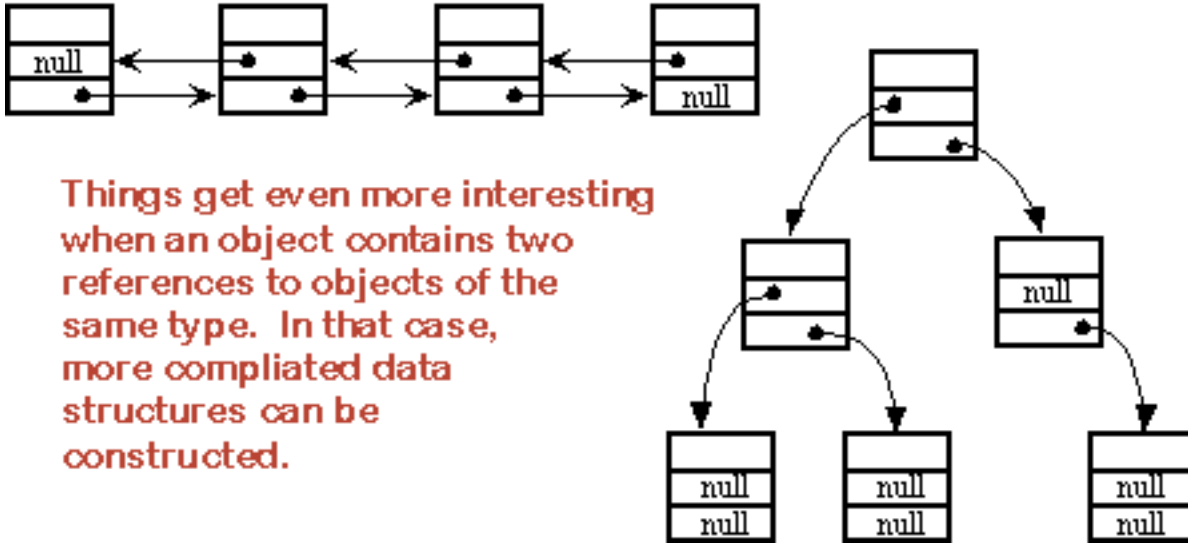
```

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 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 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 in the list refers to the next.



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.

For the rest of this section, linked lists will be constructed out of objects belonging to the class `Node` which is defined as follows:

```
class Node {
    String item;
    Node next;
}
```

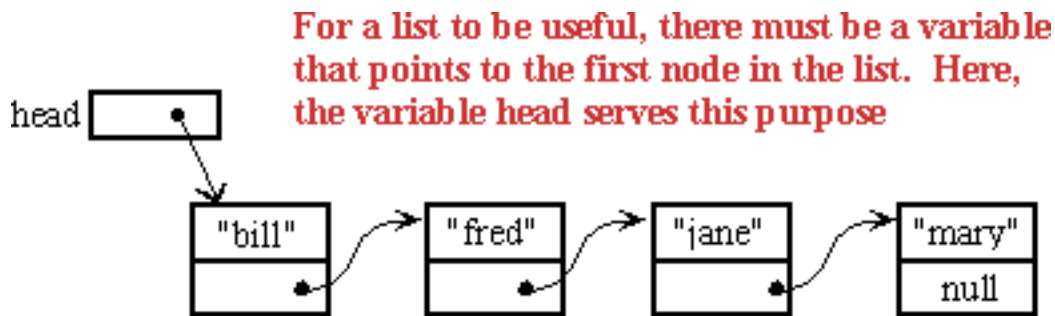
The term **node** is often used to refer to one of the objects in a linked data structure. Objects of type `Node` can be chained together as shown in the top part of the above picture. The last node in such a list can always be identified by the fact that the instance variable `next` in the last node holds the value `null` instead of a pointer to another node.

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. The subroutines are used in the following applet, which demonstrates the three types of operation. In this applet, you start with an empty list, so you have to add some strings to it before you can do anything else. The "find" operation just tells you whether a specified string is in the list.

(Applet "ListDemoConsole" would be displayed here if Java were available.)

The applet uses a class, named `StringList`, to do the list operations. An object of type `StringList` represents a linked list of `Nodes`. We'll be looking at a few aspects of this class in detail. The complete source code can be found in the file [StringList.java](#). (A standalone program that does the same thing as the applet can be found in the file [ListDemo.java](#). This program is pretty straightforward, so I won't consider it further.)

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 the sample program, an object of type `StringList` has an instance variable named `head` that serves this purpose. The variable `head` is of type `Node`, and it points to the first node in a linked list. If the list is empty, the value of `head` is `null`.



Suppose we want to know whether a specified string, `searchItem`, occurs somewhere in the list. We have to compare `searchItem` to each item in the list. To do this, we use a variable of type `Node` to "run" along the list and look at each node. Our only access to the list is through the variable `head`, so we start by getting a copy of the value in `head`:

```
Node runner = head; // Start at the first node.
```

We need a copy 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. To move from one node to the next, it is only necessary to say `runner = runner.next`. We'll know that we've reached the end of the list when `runner` becomes equal to `null`. All this is done in the instance method `find()` from the `StringList` class:

```
public boolean find(String searchItem) {
    // Returns true if the specified item is in the list, and
    // false if it is not in the list.

    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.
        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()

```

The pattern that is used in this routine occurs over and over: If head is a variable that refers to a linked list, then to process all the nodes in the list, do

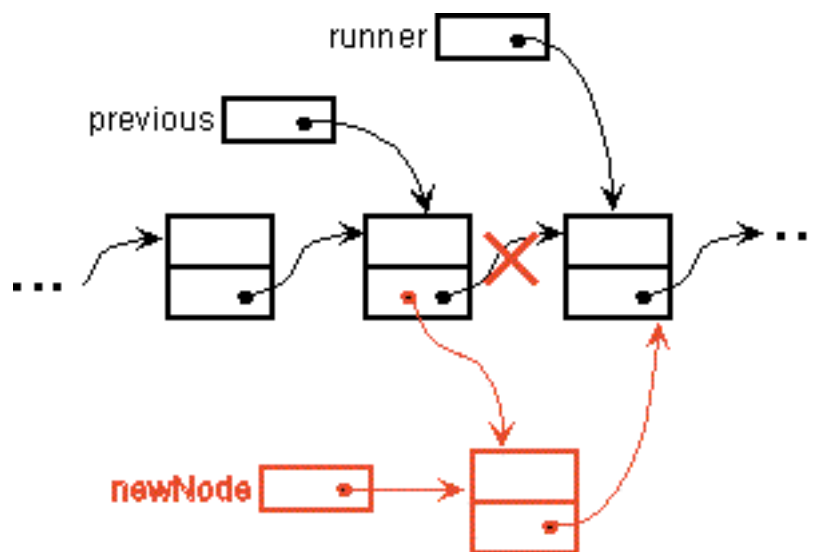
```

    runner = head;
    while ( runner != null ) {
        .
        . // Process the node that runner points to.
        .
        runner = runner.next;
    }

```

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. This is exactly what we want when the list is empty.

The problem of inserting a new item into the list is more difficult. (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.**

The command `previous.next = newNode;` can be used to make `previous.next` point to the new node, instead of to the node indicated by `runner`. And the command `newNode.next = runner` will set `newNode.next` to point to the correct place. However, before we can use these commands, we need to set up `runner` and `previous` as shown in the illustration. The idea is to start at the first node of the list, and then move along the list past all the items that are less than the new item. While doing this, we have to be aware of the danger of "falling off the end of the list." That is, we can't continue if `runner` reaches the end of the list and becomes null. If `insertItem` is the item that is to be inserted, and if we assume that it does, in fact, belong somewhere in the middle of the list, then the following code would correctly position `previous` and `runner`:

```
Node runner, previous;
previous = head;      // Start at the beginning of the list.
runner = head.next;
while ( runner != null && runner.item.compareTo(insertItem) < 0 ) {
    previous = runner; // "previous = previous.next" would also work
    runner = runner.next;
}
```

(This uses the `compareTo()` instance method from the `String` class to test whether the item in the node is less than the item that is being inserted. See [Section 2.3](#).)

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:

```
public void insert(String insertItem) {
    // Add insertItem to the list. It is allowed to add
    // multiple copies of the same item.

    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, runner 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 less 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`. This is the correct value that is needed to mark the end of the 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.

Here is the complete code for the `delete()` method:

```

public boolean delete(String deleteItem) {
    // If the specified string occurs in the list, delete it.
    // Return true if the string was found and deleted.  If the
    // string was not found in the list, return false.  (If the
    // item exists multiple times in the list, this method
    // just deletes the first one.)

    if ( head == null ) {
        // The list is empty, so it certainly
        // doesn't contain deleteItem.
        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()

```

[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Section 11.3

Stacks and Queues

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 `insert`, `delete`, and `find` operations on the list. The implementations of these operations would have been different, but their interfaces and logical behavior would still be the same.

The term **abstract data type**, or **ADT**, refers to a set of possible values and a set of operations on those values, without any specification of how the values are to be represented or how the operations are to be implemented. An "ordered list of strings" can be defined as an abstract data type. Any sequence of `Strings` that is arranged in increasing order is a possible value of this data type. The operations on the data type include inserting a new string, deleting a string, and finding a string in the list. There are often several different ways to implement the same abstract data type. For example, the "ordered list of strings" ADT can be implemented as a linked list or as an array. A program that only depends on the abstract definition of the ADT can use either implementation, interchangeably. In particular, the implementation of the ADT can be changed without affecting the program as a whole. This can make the program easier to debug and maintain, so ADT's are an important tool in software engineering.

In this section, we'll look at two common abstract data types, **stacks** and **queues**. Both stacks and queues are often implemented as linked lists, but that is not the only possible implementation. You should think of the rest of this section partly as a discussion of stacks and queues and partly as a case study in ADTs.

A stack consists of a sequence of items, which should be thought of 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.
```

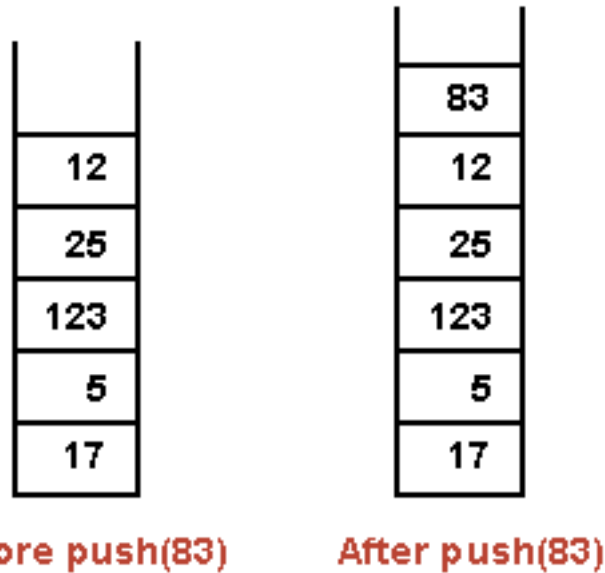
```
int pop()  -- Remove the top int from the stack and return it.
```

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 describes a "stack of ints" as an abstract data type. This ADT can be implemented in several ways, but however it is implemented, its behavior must correspond to the abstract mental image of a stack.

In a stack, all operations take place at the "top" of the stack. The "push" operation adds an item to the top of the stack. The "pop" operation removes the item on the top of the stack and returns it.



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. See [Section 5.6](#) for a discussion of nested classes. If the nesting bothers you, you could replace it with a separate Node class.)

```
public class StackOfInts {

    private static class Node {
        // An object of type Node holds one of the
        // items in the linked list that represents the stack.
        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.

    public void push( int N ) {
        // Add N to the top of the stack.
        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.
    }

    public int pop() {
        // Remove the top item from the stack, and return it.
        // Note that this routine will throw a NullPointerException
        // if an attempt is made to pop an item from an empty
        // stack. (It would be better style to define a new
        // type of Exception to throw in this case.)
        int topItem = top.item; // The item that is being popped.
        top = top.next;        // The previous second item is now on top.
        return topItem;
    }

    public boolean isEmpty() {
        // Returns true if the stack is empty. Returns false
        // if there are one or more items on the stack.
        return (top == null);
    }
} // end class StackOfInts

```

You should make sure that you understand how the push and pop operations operate on the linked list. Drawing some pictures might help. Note that the linked list is part of the private implementation of the StackOfInts class. A program that uses this class doesn't even need to know that a linked list is being used.

Now, it's pretty easy to implement a stack as an array instead of as a linked list. Since the number of items on the stack varies with time, a counter is needed to keep track of how many spaces in the array are actually in use. If this counter is called `top`, then the items on the stack are stored in positions 0, 1, ..., `top-1` in the array. The item in position 0 is on the bottom of the stack, and the item in position `top-1` is on the top of the stack. Pushing an item onto the stack is easy: Put the item in position `top` and add 1 to the value of `top`. If we don't want to put a limit on the number of items that the stack can hold, we can use the dynamic array techniques from [Section 8.3](#). Note that the typical picture of the array would show the stack "upside down", with the top of the stack at the bottom of the array. This doesn't matter. The array is just an implementation of the abstract idea of a stack, and as long as the stack operations work the way they are supposed to, we are OK. Here is a second implementation of the StackOfInts class, using a dynamic array:

```

public class StackOfInts {

    private int[] items = new int[10]; // Holds the items on the stack.

    private int top = 0; // The number of items currently on the stack.

    public void push( int N ) {
        // Add N to the top of the stack.
        if (top == items.length) {
            // The array is full, so make a new, larger array and
            // copy the current stack items into it.
            int[] newArray = new int[ 2*items.length ];
            System.arraycopy(items, 0, newArray, 0, items.length);
            items = newArray;
        }
        items[top] = N; // Put N in next available spot.
    }
}

```

```

        top++;           // Number of items goes up by one.
    }

    public int pop() {
        // Remove the top item from the stack, and return it.
        // Note that this routine will throw an
        // ArrayIndexOutOfBoundsException if an attempt is
        // made to pop an item from an empty stack.
        // (It would be better style to define a new
        // type of Exception to throw in this case.)
        int topItem = items[top - 1] // Top item in the stack.
        top--; // Number of items on the stack goes down by one.
        return topItem;
    }

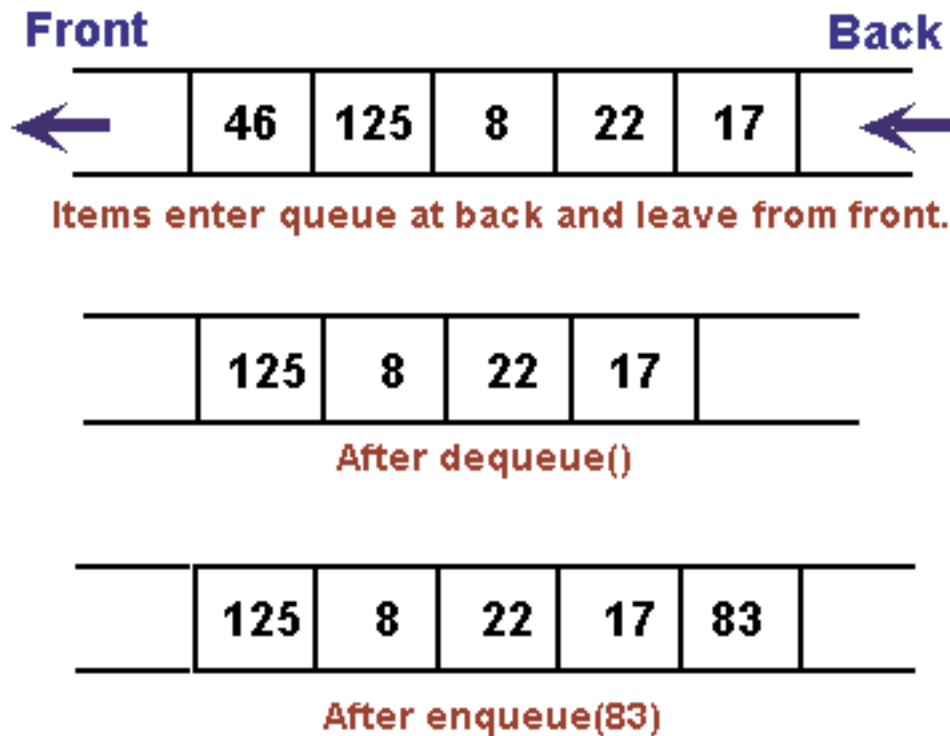
    public boolean isEmpty() {
        // Returns true if the stack is empty. Returns false
        // if there are one or more items on the stack.
        return (top == 0);
    }
} // end class StackOfInts

```

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. If a program uses one version, it should be possible to substitute the other version without changing the program. Unfortunately, though, there is one detail in which the classes behave differently: When an attempt is made to pop an item from an empty stack, the first version of the class will generate a `NullPointerException` while the second will generate an `ArrayIndexOutOfBoundsException`. It would be better to define a new `EmptyStackException` class and use it in both versions. In fact, the original description of the "stack of ints" ADT should have specified exactly what happens when an attempt is made to pop an item from an empty stack. This is just the sort of small detail that is often left out of interface specifications, causing no end of problems!

Queues are similar to stacks in that a queue consists of a sequence of items, and there are restrictions about how items can be added to and removed from the list. However, a queue has two ends, called the front and the back of the queue. Items are always added to the queue at the back and removed from the queue at the front. The operations of adding and removing items are called **enqueue** and **dequeue**. An item that is added to the back of the queue will remain on the queue until all the items in front of it have been removed. This should sound familiar. A queue is like a "line" or "queue" of customers waiting for service. Customers are serviced in the order in which they arrive on the queue.

In a queue, all operations take place at one end of the queue or the other. The "enqueue" operation adds an item to the "back" of the queue. The "dequeue" operation removes the item at the "front" of the queue and returns it.



A queue can hold items of any type. For a queue of `ints`, the enqueue and dequeue operations can be implemented as instance methods in a "QueueOfInts" class. We also need an instance method for checking whether the queue is empty:

```
void enqueue(int N)  -- Add N to the back of the queue.
```

```
int dequeue()  -- Remove the item at the front and return it.
```

```
boolean isEmpty()  -- Return true if the queue is empty.
```

A queue can be implemented as a linked list or as an array. An efficient array implementation is a little trickier than the array implementation of a stack, so I won't give it here. In the linked list implementation, the first item of the list is 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 on 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;
```



```

    }
    // At this point, tail.next is null, so tail points to
    // the last node in the list.

```

However, it would be very inefficient to do this over and over every time an item is enqueued. For the sake of efficiency, we'll keep a pointer to the last node in an instance variable. We just 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 very difficult:

```

public class QueueOfInts {

    private static class Node {
        // An object of type Node holds one of the items
        // in the linked list that represents the queue.
        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.

    void enqueue( int N ) {
        // Add N to the back of the queue.
        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;
        }
    }

    int dequeue() {
        // Remove and return the front item in the queue.
        // Note that this can throw a NullPointerException.
        int firstItem = head.item;
        head = head.next; // The previous second item is now first.
        if (head == null) {
            // The queue has become empty. The Node that was
            // deleted was the tail as well as the head of the
            // list, so now there is no tail. (Actually, the
            // class would work fine without this step.)
            tail = null;
        }
        return firstItem;
    }
}

```

```

    }

    boolean isEmpty() {
        // Return true if the queue is empty.
        return (head == null);
    }

} // end class QueueOfInts

```

Queues are typically used in a computer (as in real life) when only one item can be processed at a time, but several items can be waiting for processing. For example:

- In a Java program that has multiple threads, the threads that want processing time on the CPU are kept in a queue. When a new thread is started, it is added to the back of the queue. A thread is removed from the front of the queue, given some processing time, and then -- if it has not terminated -- is sent to the back of the queue to wait for another turn.
- Events such as keystrokes and mouse clicks are stored in a queue called the "event queue". A program removes events from the event queue and processes them. It's possible for several more events to occur while one event is being processed, but since the events are stored in a queue, they will always be processed in the order in which they occurred.
- A `ServerSocket`, as covered in [Section 10.4](#), has an associated queue which contains connection requests that have been received but not yet serviced. The `ServerSocket`'s `accept()` method gets the next connection request from the front of this queue.

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 applet shown below. When you click on a white square in the grid, the applet will gradually mark all the squares in the grid, starting from the one where you click. To understand how the applet does this, think of yourself in the place of the applet. 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, take 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 an index card and put the card in the pile.

While a square is in the pile, waiting to be processed, it is colored red. When a square is taken from the pile and processed, its color changes to gray. Eventually, all the squares have been processed and the procedure ends. In the index card analogy, the pile of cards contains all the red squares.

The applet 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 the cards in the pile. The order of processing is very different in these three cases.

You should experiment with the applet to see how it all works. Try to understand how stacks and queues are being used. Try starting from one of the corner squares. While the process is going on, you can click on other white squares, and they will be added to the pile. When you do this with a stack, you should notice that the square you click is processed immediately, and all the red squares that were already waiting for processing have to wait. On the other hand, if you do this with a queue, the square that you click will wait

its turn. The source code for this applet can be found in the file [DepthBreadth.java](#).

(Applet "DepthBreadth" would be displayed here
if Java were available.)

Queues seem very natural because they occur so often in real life, but there are times when stacks are appropriate and even essential. For example, consider what happens when a routine calls a subroutine. The first routine is suspended while the subroutine is executed, and it will continue only when the subroutine returns. Now, suppose that the subroutine calls a second subroutine, and the second subroutine calls a third, and so on. Each subroutine is suspended while the subsequent subroutines are executed. The computer has to keep track of all the subroutines that are suspended. It does this with a stack.

When a subroutine is called, an **activation record** is created for that subroutine. The activation record contains information relevant to the execution of the subroutine, such as its local variables and parameters. The activation record for the subroutine is placed on a stack. It will be removed from the stack and destroyed when the subroutine returns. If the subroutine calls another subroutine, the activation record of the second subroutine is pushed onto the stack, on top of the activation record of the first subroutine. The stack can continue to grow as more subroutines are called, and it shrinks as those subroutines return.

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 will be used for later processing, so we push it onto the stack, on top of the 2, which 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
    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](#), which 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 used in the program is defined in the file [NumberStack.java](#). The NumberStack 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.

Here is an applet that simulates the PostfixEval program:

(Applet "PostfixEvalConsole" would be displayed here
if Java were available.)

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:

```

static void readAndEvaluate() {
    // 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. (This is checked in the
    // main() routine.)

    NumberStack stack; // For evaluating the expression.

    stack = new NumberStack(); // Make a new, empty stack.

    TextIO.putln();

    while (TextIO.peek() != '\n') {

        if ( Character.isDigit(TextIO.peek()) ) {
            // The next item in input is a number. Read it and
            // save it on the stack.
            double num = TextIO.getDouble();
            stack.push(num);
            TextIO.putln("    Pushed constant " + num);
        }
        else {
            // Since the next item is not a number, the only thing

```

```

        // it can legally be is an operator.  Get the operator
        // and perform the operation.
char op; // The operator, which must be +, -, *, /, or ^.
double x,y; // The operands, from the stack.
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 legal operations.
    TextIO.putln("\nIllegal operator found in input: " + op);
    return;
}
if (stack.isEmpty()) {
    TextIO.putln(
        "    Stack is empty while trying to evaluate " + op);
    TextIO.putln("\nNot enough numbers in expression!");
    return;
}
y = stack.pop();
if (stack.isEmpty()) {
    TextIO.putln(
        "    Stack is empty while trying to evaluate " + op);
    TextIO.putln("\nNot enough numbers in expression!");
    return;
}
x = stack.pop();
switch (op) {
    case '+': answer = x + y; break;
    case '-': answer = x - y; break;
    case '*': answer = x * y; break;
    case '/': answer = x / y; break;
    default: answer = Math.pow(x,y); // (op must be '^'.)
}
stack.push(answer);
TextIO.putln("    Evaluated " + op + " and pushed " + answer);
}

skipSpaces(); // Skips past any blanks in the input, before
              // going back to the start of the while loop to
              // test TextIO.peek() again.

} // 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 input is really non-empty.
    TextIO.putln("No expression provided.");
    return;
}

double value = stack.pop(); // Value of the expression.
TextIO.putln("    Popped " + value + " at end of expression.");

if (stack.isEmpty() == false) {

```

```
        TextIO.putln("    Stack is not empty.");
        TextIO.putln("\nNot enough operators for all the numbers!");
        return;
    }

    TextIO.putln("\nValue = " + value);

} // end readAndEvaluate()
```

Postfix expressions are often used internally by computers. In fact, the Java virtual machine is a "stack machine" which uses the stack-based approach to expression evaluation that we have been discussing. The algorithm can easily be extended to handle variables, as well as constants. When a variable is encountered in the expression, the value of the variable is pushed onto the stack. It also works for operators with more or fewer than two operands. As many operands as are needed are popped from the stack and the result is pushed back on to 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.

[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Section 11.4

Binary Trees

WE HAVE SEEN in the two previous sections how objects can be linked into lists. When an object contains two pointers to objects of the same type, structures can be created that are much more complicated than linked lists. In this section, we'll look at one of the most basic and useful structures of this type: **binary trees**. Each of the objects in a binary tree contains two pointers, typically called `left` and `right`. In addition to these pointers, of course, the nodes can contain other types of data. For example, a binary tree of integers could be made up of objects of the following type:

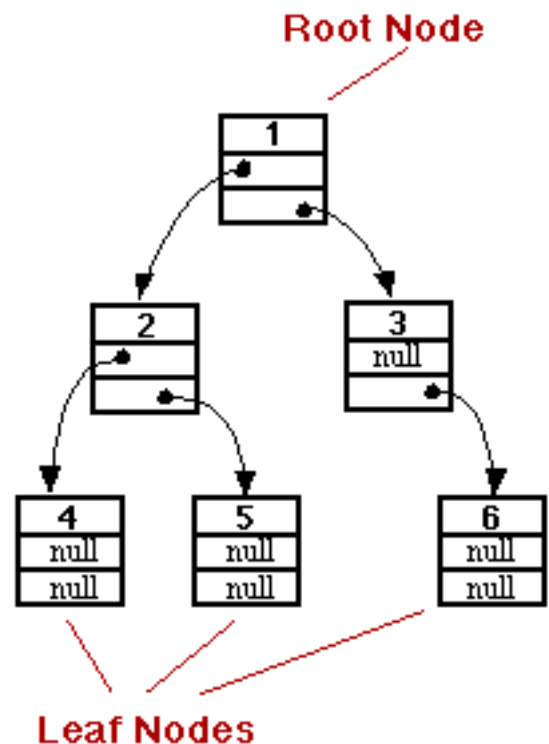
```
class TreeNode {
    int item;           // The data in this node.
    TreeNode left;      // Pointer to the left subtree.
    TreeNode right;     // Pointer to the right subtree.
}
```

The `left` and `right` pointers in a `TreeNode` can be `null` or can point to other objects of type `TreeNode`. A node that points to another node is said to be the **parent** of that node, and the node it points to is called a **child**. In the picture at the right, 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 with the analogy to real trees. But at least you can see the branching, tree-like structure that gives a binary tree its name.

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. The heart of 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:

```
static int countNodes( TreeNode root ) {
    // Count the nodes in the binary tree to which
    // root points, and return the answer.
    if ( root == null )
        return 0; // The tree is empty. It contains no nodes.
    else {
        int count = 1; // Start by counting the root.
        count += countNodes(root.left); // Add the number of nodes
                                         // in the left subtree.
        count += countNodes(root.right); // Add the number of nodes
                                         // in the right subtree.
        return count; // Return the total.
    }
} // end countNodes()
```

Or, consider the problem of printing the items in a binary tree. If the tree is empty, there is nothing to do. If the tree is non-empty, then it consists of a root and two subtrees. Print the item in the root and use recursion to print the items in the subtrees. Here is a subroutine that prints all the items on one line of output:

```
static void preorderPrint( TreeNode root ) {
    // 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.
    if ( root != null ) { // (Otherwise, there's nothing to print.)
        System.out.print( root.item + " " ); // Print the root item.
        preorderPrint( root.left ); // Print items in left subtree.
        preorderPrint( root.right ); // Print items in right subtree.
    }
} // end preorderPrint()
```

This routine is called "preorderPrint" because it uses a **preorder traversal** of the tree. In a preorder traversal, the root node of the tree is processed first, then the left subtree is traversed, then the right subtree. In a **postorder traversal**, the left subtree is traversed, then the right subtree, and then the root node is processed. And in an **inorder traversal**, the left subtree is traversed first, then the root node is processed, then the right subtree is traversed. Printing subroutines that use postorder and inorder traversal differ from preorderPrint only in the placement of the statement that outputs the root item:

```
static void postorderPrint( TreeNode root ) {
    // 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.
    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()
```

```

static void inorderPrint( TreeNode root ) {
    // 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, followed by the items in
    // the right subtree.
    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

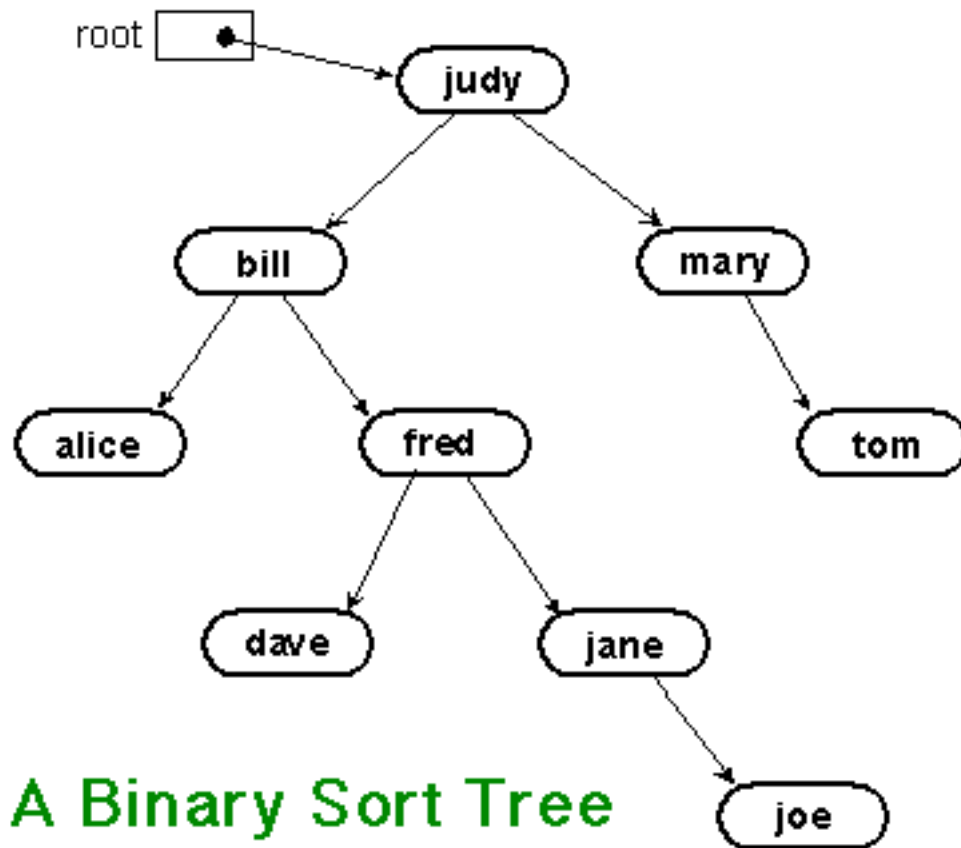
```

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.

Binary Sort Trees

One of the examples in [Section 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. (See [Section 8.4](#).) However, inserting a new item into the array is still inefficient since it means moving, on average, half of the items in the array to make a space for the new item. A binary tree can be used to store an ordered list of strings, or other items, in a way that makes both searching and insertion efficient. A binary tree used in this way is called a **binary sort tree**.

A binary sort tree is a binary tree with the following property: For every node in the tree, the item in that node is greater than every item in the left subtree of that node, and it is less than or equal to all the items in the right subtree of that node. Here for example is a binary sort tree containing items of type `String`. (In this picture, I haven't bothered to draw all the pointer variables. Non-null pointers are shown as arrows.)



Binary sort trees have this useful property: An inorder traversal of the tree will process the items in increasing order. In fact, this is really just another way of expressing the definition. For example, if an inorder traversal is used to print the items in the tree shown above, then the items will be in alphabetical order. The definition of an inorder traversal guarantees that all the items in the left subtree of "judy" are printed before "judy", and all the items in the right subtree of "judy" are printed after "judy". But the binary sort tree property guarantees that the items in the left subtree of "judy" are precisely those that precede "judy" in alphabetical order, and all the items in the right subtree follow "judy" in alphabetical order. So, we know that "judy" is output in its proper alphabetical position. But the same argument applies to the subtrees. "Bill" will be output after "alice" and before "fred" and its descendents. "Fred" will be output after "dave" and before "jane" and "joe". And so on.

Suppose that we want to search for a given item in a binary search tree. Compare that item to the root item of the tree. If they are equal, we're done. If the item we are looking for is less than the root item, then we need to search the left subtree of the root -- the right subtree can be eliminated because it only contains items that are greater than or equal to the root. Similarly, if the item we are looking for is greater than the item in the root, then we only need to look in the right subtree. In either case, the same procedure can then be applied to search the subtree. Inserting a new item is similar: Start by searching the tree for the position where the new item belongs. When that position is found, create a new node and attach it to the tree at that position.

Searching and inserting are efficient operations on a binary search tree, provided that the tree is close to being **balanced**. A binary tree is balanced if for each node, the left subtree of that node contains approximately the same number of nodes as the right subtree. In a perfectly balanced tree, the two numbers differ by at most one. Not all binary trees are balanced, but if the tree is created randomly, there is a high probability that the tree is approximately balanced. 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 from [Section 8.4](#), and the result is a similarly efficient algorithm.

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. Here is an applet that simulates this program:

(Applet "SortTreeConsole" would be displayed here
if Java were available.)

In this program, nodes in the binary tree are represented using the following class, including a simple constructor that makes creating nodes easier:

```
class TreeNode {
    // An object of type TreeNode represents one node
    // in a binary tree of strings.
    String item;        // The data in this node.
    TreeNode left;      // Pointer to left subtree.
    TreeNode right;     // Pointer to right subtree.
    TreeNode(String str) {
        // Constructor.  Make a node containing str.
        item = str;
    }
} // end class TreeNode
```

A static member variable of type `TreeNode` points to the binary sort tree that is used by the program:

```
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:

```
static boolean treeContains( TreeNode node, String item ) {
    // Return true if item is one of the items in the binary
    // sort tree to which node points.  Return false if not.
    if ( node == null ) {
        // Tree is empty, so it certainly doesn't contain item.
        return false;
    }
    else if ( item.equals(node.item) ) {
        // Yes, the item has been found in the root node.
        return true;
    }
    else if ( item.compareTo(node.item) < 0 ) {
        // If the item occurs, it must be in the left subtree.
        // So, return the result of searching the left subtree.
        return treeContains( node.left, item );
    }
    else {
        // If the item occurs, it must be in the right subtree.
        // So, return the result of searching the right subtree.
        return treeContains( node.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 just follows the rule: 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 non-recursive version of the search routine:

```
static boolean treeContainsNR( TreeNode root, String item ) {
    // Return true if item is one of the items in the binary
    // sort tree to which root points.  Return false if not.
    TreeNode runner; // For "running" down the tree.
    runner = root;   // Start at the root node.
    while (true) {
        if (runner == null) {
            // We've fallen off the tree without finding item.
            return false;
        }
        else if ( item.equals(node.item) ) {
            // We've found the item.
            return true;
        }
        else if ( item.compareTo(node.item) < 0 ) {
            // If the item occurs, it must be in the left subtree,
            // So, advance the runner down one level to the left.
            runner = runner.left;
        }
        else {
            // If the item occurs, it must be in the right subtree.
            // So, advance the runner down one level to the right.
            runner = runner.right;
        }
    } // end while
} // end treeContainsNR();
```

The subroutine for inserting a new item into the tree turns out to be more similar to the non-recursive search routine than to the recursive. The insertion routine has to handle the case where the tree is empty. In that case, the value of `root` must be changed to point to a node that contains the new item:

```
root = new TreeNode( newItem );
```

But this means, effectively, that the `root` can't be passed as a parameter to the subroutine, because it is impossible for a subroutine to change the value stored in an actual parameter. (I should note that this is something that **is possible in other languages.**) **Recursion uses parameters in an essential way. There are ways to work around the problem, but the easiest thing is just to use a non-recursive insertion routine that accesses the static member variable `root` directly.** One difference between inserting an item and searching for an item is that we have to be careful not to fall off the tree. That is, we have to stop searching just before `runner` becomes `null`. When we get to an empty spot in the tree, that's where we have to insert the new node:

```
static void treeInsert(String newItem) {
    // 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.)
    if ( root == null ) {
```

```

        // The tree is empty. Set root to point to a new node
        // containing the new item.
        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 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()

```

Expression Trees

Another application of trees is to store mathematical expressions such as $15 * (x + y)$ or $\text{sqrt}(42) + 7$ 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**.

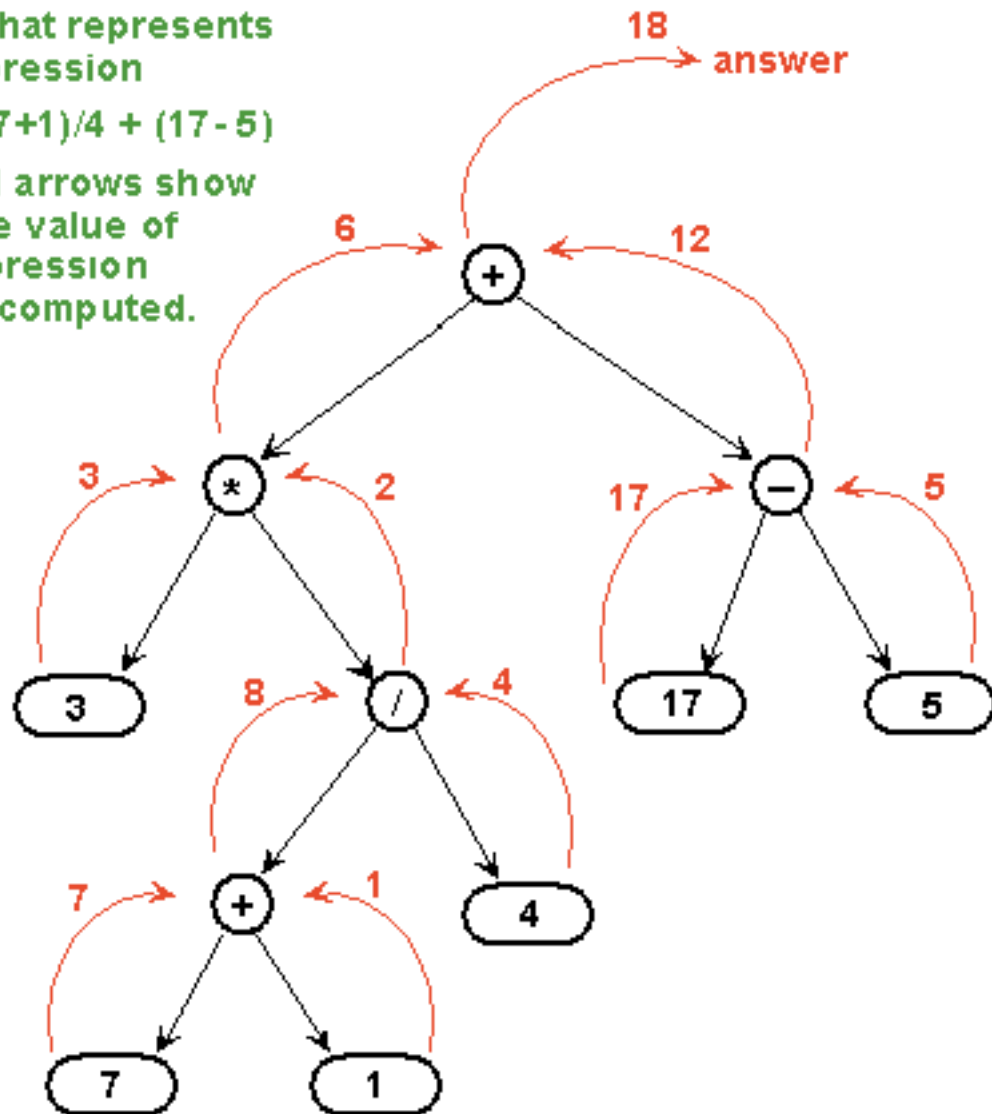
Given an expression tree, it's easy to find the value of the expression that it represents. Each node in the tree has an associated value. If the node is a leaf node, then its value is simply the number that the node contains. If the node contains an operator, then the associated value is computed by first finding the values of its child nodes and then applying the operator to those values. The process is shown by the red 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.

A tree that represents the expression

$$3 * (7+1)/4 + (17-5)$$

The red arrows show how the value of the expression can be computed.



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:

```
class ExpNode { // A node in an expression tree.

    static final int NUMBER = 0, // Possible values for kind.
                  OPERATOR = 1;

    int 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 = NUMBER;
        number = val;
    }

    ExpNode( char op, ExpNode left, ExpNode right ) {
        // Constructor for making a node of type OPERATOR.
        kind = 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 == NUMBER ) {
        // The value of a NUMBER node is the number it holds.
        return node.number;
    }
    else { // The kind must be OPERATOR.
        // Get the values of the operands and combine them
        // using the operator.
        double leftVal = getValue( node.left );
        double rightVal = getValue( node.right );
        switch ( node.op ) {
            case '+': return leftVal + rightVal;
            case '-': return leftVal - rightVal;
            case '*': return leftVal * rightVal;
            case '/': return leftVal / rightVal;
            default: return Double.NaN; // Bad operator.
        }
    }
} // end getValue()

```

Although this approach works, a more object-oriented approach is to note that since there are two types of nodes, there should be two classes to represent them, ConstNode and BinOpNode. To represent the general idea of a node in an expression tree, we need another class, ExpNode. Both ConstNode and BinOpNode will be subclasses of ExpNode. Since any actual node will be either a ConstNode or a BinOpNode, ExpNode should be an abstract class. (See [Section 5.4](#).) Since one of the things we want to do with nodes is find their values, each class should have an instance method for finding the value:

```

abstract class ExpNode {
    // Represents a node of any type in an expression tree.

    abstract double value(); // Return the value of this node.
}

```

```

    } // end class ExpNode

class ConstNode extends ExpNode {
    // Represents a node that holds a number.

    double number; // The number in the node.

    ConstNode( double val ) {
        // Constructor. Create a node to hold val.
        number = val;
    }

    double value() {
        // The value is just the number that the node holds.
        return number;
    }

} // end class ConstNode

class BinOpNode extends ExpNode {
    // Represents a node that holds an operator.

    char op; // The operator.
    ExpNode left; // The left operand.
    ExpNode right; // The right operand.

    BinOpNode( char op, ExpNode left, ExpNode right ) {
        // Constructor. Create a node to hold the given data.
        this.op = op;
        this.left = left;
        this.right = right;
    }

    double value() {
        // To get the value, compute the value of the left and
        // right operands, and combine them with the operator.
        double leftVal = left.value();
        double rightVal = right.value();
        switch ( op ) {
            case '+': return leftVal + rightVal;
            case '-': return leftVal - rightVal;
            case '*': return leftVal * rightVal;
            case '/': return leftVal / rightVal;
            default: return Double.NaN; // Bad operator.
        }
    }

} // end class BinOpNode

```

Note that the left and right operands of a BinOpNode are of type ExpNode, not BinOpNode. This allows the operand to be either a ConstNode or another BinOpNode -- or any other type of ExpNode that we might eventually create. Since every ExpNode has a value() method, we can call left.value() to compute the value of the left operand. If left is in fact a ConstNode, this will call

the `value()` method in the `ConstNode` class. If it is in fact a `BinOpNode`, then `left.value()` will call the `value()` method in the `BinOpNode` class. Each node knows how to compute its own value.

Although it might seem more complicated at first, the object-oriented approach has some advantages. For one thing, it doesn't waste memory. In the original `ExpNode` class, only some of the instance variables in each node were actually used, and we needed an extra instance variable to keep track of the type of node. More important, though, is the fact that new types of nodes can be added more cleanly, since it can be done by creating a new subclass of `ExpNode` rather than by modifying an existing class.

We'll return to the topic of expression trees in the next section, where we'll see how to create an expression tree to represent a given expression.

[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Section 11.5

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, after all, 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?

Natural and artificial languages are similar in that they have a structure known as grammar or syntax. Syntax can be expressed by a set of rules that describe what it means to be a legal sentence or program. For programming languages, syntax rules are often expressed in **BNF** (Backus-Naur Form), a system that was developed by computer scientists John Backus and Peter Naur in the late 1950s. Interestingly, an equivalent system was developed independently at about the same time by linguist Noam Chomsky to describe the grammar of natural language. BNF cannot express all possible syntax rules. For example, it can't express the fact that a variable must be defined before it is used. Furthermore, it says nothing about the meaning or semantics of the language. The problem of specifying the semantics of a language -- even of an artificial programming language -- is one that is still far from being completely solved. However, BNF does express the basic structure of the language, and it plays a central role in the design of translation programs.

In English, terms such as "noun", "transitive verb," and "propositional 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 it can be 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 one or more 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> [ <propositional-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 `<propositional-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.

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 to determining the meaning of the phrase -- which for a programming language means translating it into machine language. Although we will look at only a simple example, I hope it will be enough to convince you that compilers can in fact be written and understood by mortals and to give you some idea of how that can be done.

The parsing method that we will use is called **recursive descent parsing**. It is not the only possible parsing method, or the most efficient, but it is the one most suited for writing compilers by hand (rather than with the help of so called "parser generator" programs). In a recursive descent parser, every rule of the BNF grammar is the model for a subroutine. Not every BNF grammar is suitable for recursive descent parsing. The grammar must satisfy a certain property. Essentially, while parsing a phrase, it must be possible to tell what syntactic category is coming up next just by looking at the next item in the input. Many grammars are designed with this property in mind.

I should also mention that many variations of BNF are in use. The one that I've described here is one that is well-suited for recursive descent parsing.

When we try to parse a phrase that contains a syntax error, we need some way to respond to the error. A convenient way of doing this is to throw an exception. I'll use an exception class called `ParseError`, defined as follows:

```
class ParseError extends Exception {
    // Represents a syntax error detected while parsing.
    ParseError(String message) {
        // Construct a ParseError object containing the
```

```

        // given string as its error message.
        super(message); // (Call constructor from superclass.)
    }
}

```

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 following routine before trying to look ahead to see what's coming up next in input:

```

static void skipBlanks() {
    // Skip over blanks and tabs in standard input.
    while ( TextIO.peek() == ' ' || TextIO.peek() == '\t' )
        TextIO.getAnyChar();
}

```

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 positive 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.

```

static char getOperator() throws ParseError {
    // If the next character in input is one of the legal operators,
    // read it and return it. Otherwise, throw a ParseError.
    skipBlanks(); // Skip past any blanks and tabs.
    char op = TextIO.peek(); // Look ahead at the next character.
    if ( op == '+' || op == '-' || op == '*' || op == '/' ) {
        TextIO.getAnyChar(); // Read the character.
        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 `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.

```
static double expressionValue() throws ParseError {
    // Read an expression from the current line of input and
    // return its value.
    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(); // First expression.
        char op = getOperator(); // The operator.
        double rightVal = expressionValue(); // Second expression.
        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; // (Actually, this can't occur.)
        }
    }
    else {
        throw new ParseError("Encountered unexpected character, \"" +
                               TextIO.peek() + "\" in input.");
    }
} // end expressionValue()
```

I hope that you can see how this routine corresponds to the BNF rule. Where the rule uses "|" to give a choice between alternatives, there is an if statement in the routine to determine which choice to take. Where the rule contains a sequence of items, "(" <expression> <operator> <expression> ")", there is a sequence of statements in the subroutine to read each item in turn.

When `expressionValue()` is called to evaluate the expression $((34-17)*8)+(2*7)$, it sees the "(" at the beginning of the input, so the else part of the if statement is executed. The "(" is read. Then the first recursive call to `expressionValue()` reads and evaluates the subexpression $(34-17)*8$, the call to `getOperator()` reads the "+" operator, and the second recursive call to `expressionValue()` reads and evaluates the second subexpression $2*7$. Finally, the ")" at the end of the expression is read. Of course, reading the first subexpression, $(34-17)*8$, involves further recursive calls to the `expressionValue()` routine, but it's better not to think too deeply about that! Rely on the recursion to

handle the details.

You'll find a complete program that uses these routines in the file [SimpleParser1.java](#).

Fully parenthesized expressions aren't very natural for people to use. But with ordinary expressions, we have to worry about the question of operator precedence, which tells us, for example, that the "*" in the expression "5+3*7" is applied before the "+". The complex expression "3*6+8*(7+1)/4-24" should be seen as made up of three "terms", 3*6, 8*(7+1)/4, and 24, combined with "+" and "-" operators. A term, on the other hand, can be made up of several factors combined with "*" and "/" operators. For example, 8*(7+1)/4 contains the factors 8, (7+1) and 4. This example also shows that a factor can be either a number or an expression in parentheses. To complicate things a bit more, we allow for leading minus signs in expressions, as in "-(3+4)" or "-7". (Since a <number> is a positive number, this is the only way we can get negative numbers. It's done this way to avoid "3 * -7", for example.) This structure can be expressed by the BNF rules

```
<expression> ::= [ "-" ] <term> [ ( "+" | "-" ) <term> ]...
<term> ::= <factor> [ ( "*" | "/" ) <factor> ]...
<factor> ::= <number> | "(" <expression> ")"
```

The first rule uses the "[]..." notation, which says that the items that it encloses can occur zero, one, two, or more times. This means that an <expression> can begin, optionally, with a "-". Then there must be a <term> which can optionally be followed by one of the operators "+" or "-" and another <term>, optionally followed by another operator and <term>, and so on. In a subroutine that reads and evaluates expressions, this repetition is handled by a while loop. An if statement is used at the beginning of the loop to test whether a leading minus sign is present:

```
static double expressionValue() throws ParseError {
    // Read an expression from the current line of input and
    // return its value.
    skipBlanks();
    boolean negative; // True if there is a leading minus sign.
    negative = false;
    if (TextIO.peek() == '-') {
        TextIO.getAnyChar(); // Read the "-"
        negative = true;
    }
    double val; // Value of the expression.
    val = termValue(); // Get the value of the first term.
    if (negative)
        val = -val; // Apply the leading "-" operator.
    skipBlanks();
    while ( TextIO.peek() == '+' || TextIO.peek() == '-' ) {
        // There is a "+" or "-" followed by another term.
        // 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 the next term.
        if (op == '+')
            val += nextVal;
        else
            val -= nextVal;
        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](#).

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. Here is an applet that simulates the program:

(Applet "SimpleParser3Console" would be displayed here
if Java were available.)

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 the [previous section](#), 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 the previous section. Another class, UnaryMinusNode, has been introduced to represent the unary minus operation. I've added a method, printStackCommands(), to each class. This method is responsible for printing out the stack operations that are necessary to evaluate an expression. Here for example is the new BinOpNode class from [SimpleParser3.java](#):

```
class BinOpNode extends ExpNode {
    // An expression node representing a binary operator.

    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.
        this.op = op;
        this.left = left;
        this.right = right;
    }

    double value() {
        // The value is obtained by evaluating the left and right
        // operands and combining the values with the operator.
        double x = left.value();
        double y = right.value();
        switch (op) {
            case '+': return x + y;
            case '-': return x - y;
            case '*': return x * y;
            case '/': return x / y;
            default:  return Double.NaN; // Bad operator!
        }
    }
}
```

```

void printStackCommands() {
    // To evaluate the expression on a stack machine, first do
    // whatever is necessary to evaluate the left operand,
    // leaving the answer on the stack. Then do the same thing
    // for the right operand. Then apply the operator (which
    // means popping the operands, applying the operator, and
    // pushing the result).
    left.printStackCommands();
    right.printStackCommands();
    TextIO.putln(" Operator " + op);
}

} // end class BinOpNode

```

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.
    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);
    }
    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);
        skipBlanks();
    }
    return exp;
} // end expressionTree()

```

In some real compilers, the parser creates a tree to represent the program that is being parsed. This tree is called a **parse tree**. Parse trees are somewhat different in form from expression trees, but the purpose is the same. Once you have the tree, there are a number of things you can do with it. For one thing, it can be used

to generate machine language code. But there are also techniques for examining the tree and detecting certain types of programming errors, such as an attempt to reference a local variable before it has been assigned a value. (The Java compiler, of course, will reject the program if it contains such an error.) It's also possible to manipulate the tree to **optimize** the program. In optimization, the tree is transformed to make the program more efficient before the code is generated.

And so we wind up 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.

End of Chapter 11

[[Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Programming Exercises For Chapter 11

THIS PAGE CONTAINS programming exercises based on material from [Chapter 11](#) of this [on-line Java textbook](#). Each exercise has a link to a discussion of one possible solution of that exercise.

Exercise 11.1: The `DirectoryList` program, given as an example at the end of [Section 10.2](#), 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: Make a new version of the sample program [WordList.java](#), from [Section 10.3](#), that stores words in a binary sort tree instead of in an array.

[See the solution!](#)

Exercise 11.3: 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 11.4: [Section 11.4](#) 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 `TreeNode`s, so you will need to write a class to represent such queues.

[See the solution!](#)

Exercise 11.5: In [Section 11.4](#), I say that "if the [binary sort] tree is created randomly, 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 `insert()` 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 the routine recursively, the parameter increases by 1.

[See the solution!](#)

Exercise 11.6: The parsing programs in Section 11.5 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 11.7: This exercise builds on the previous exercise, Exercise 11.6. 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
 1. The derivative of $A+B$ is $dA+dB$.
 2. The derivative of $A-B$ is $dA-dB$.
 3. The derivative of $A*B$ is $A*dB + B*dA$.
 4. 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, you should print out the stack operations that define the derivative, instead of the operations for the original expression. 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.

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. 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 subroutines that we have written to process trees, since it would lead to infinite recursion.)

[See the solution!](#)

Quiz Questions For Chapter 11

THIS PAGE CONTAINS A SAMPLE quiz on material from [Chapter 11](#) of this [on-line Java textbook](#). You should be able to answer these questions after studying that chapter. Sample answers to all the quiz questions can be found [here](#).

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.println("]");
    }
}
```

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 find the sum of all the `ints` in a linked list. The subroutine should have a parameter of type `ListNode` and should return a value of type `int`.

Question 4: What are the three operations on a *stack*?

Question 5: What is the basic difference between a stack and a queue?

Question 6: What is an *activation record*? What role does a stack of activation records play in a computer?

Question 7: Suppose that a binary tree 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 8: What is a *postorder traversal* of a binary tree?

Question 9: Suppose that a `<multilist>` is defined by the BNF rule

`<multilist> ::= <word> | "(" [<multilist>]... "`

where a `<word>` can be any sequence of letters. Give five different `<multilist>`'s that can be generated by this rule. (This rule, by the way, is almost the entire syntax of the programming language LISP! LISP is known for its simple syntax and its elegant and powerful semantics.)

Question 10: Explaining what is meant by *parsing* a computer program.

[[Answers](#) | [Chapter Index](#) | [Main Index](#)]

Chapter 12

Generic Programming and Collection Classes

HOW TO AVOID REINVENTING the wheel? Many data structures and algorithms, such as those from the previous chapter, 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.

Contents of Chapter 12:

- Section 1: [Generic Programming](#)
 - Section 2: [List and Set Classes](#)
 - Section 3: [Map Classes](#)
 - Section 4: [Programming with Collection Classes](#)
 - [Programming Exercises](#)
 - [Quiz on this Chapter](#)
-

[[First Section](#) | [Previous Chapter](#) | [Main Index](#)]

Section 12.1

Generic Programming

GENERIC PROGRAMMING refers to writing code that will work for many types of data. We encountered the term in [Section 8.3](#), where we looked at dynamic arrays of integers. The source code presented there for working with dynamic arrays of integers works only for data of type `int`. But the source code for dynamic arrays of `double`, `String`, `JButton`, or any other type would be almost identical. It seems silly to write essentially the same code over and over. As we saw in Section 8.3, Java goes some distance towards solving this problem by providing the `ArrayList` class. An `ArrayList` is essentially a dynamic array of values of type `Object`. Since every class is a sub-class of `Object`, objects belonging to any class can be stored in an `ArrayList`. This is an example of generic programming: The source code for the `ArrayList` class was written once, and it works for objects of any type. (It does, not, however, work for data belonging to the primitive types, such as `int` and `double`.)

The `ArrayList` class is just one of several classes and interfaces that are used for generic programming in Java. We will spend this chapter looking at these classes and how they are used. All the classes discussed in this chapter are defined in the package `java.util`, and you will need an import statement at the beginning of your program to get access to them. (Before you start putting `import java.util.*` at the beginning of every program, you should know that some things in `java.util` have names that are the same as things in other packages. For example, both `java.util.List` and `java.awt.List` exist.)

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, but in the context of the overall design of Java, it might be close to optimal. To get some perspective on generic programming in general, it might be useful to look very briefly at generic programming in two other languages.

Generic Programming in Smalltalk

Smalltalk was one of the very first object-oriented programming languages. It is still used today. Although it has not achieved anything like the popularity of Java or C++, 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 9](#).) 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 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.

Generic Programming in C++

Unlike Smalltalk, C++ is a very strongly typed language, even more so than Java. Every variable has a type, and can only hold data values of that type. This means that the type of generic programming used in Smalltalk is impossible. Furthermore, C++ does not have anything corresponding to Java's `Object` class. That is, there is no class that is a superclass of all other classes. This means that C++ can't use Java's style of generic programming either. Nevertheless, C++ has a powerful and flexible system of generic programming. It is made possible by a language feature known as **templates**. In C++, instead of writing a different sorting subroutine for each type of data, you can write a single subroutine template. The template is not a subroutine; it's more like a factory for making subroutines. We can look at an example, since the syntax of C++ is very similar to Java's:

```
template<class ItemType>
void sort( ItemType A[], int count ) {
    // Sort count items in the array, A, into increasing order.
    // The algorithm that is used here is selection sort.
    for (int i = count-1; i > 0; i--) {
        int position_of_max = 0;
        for (int j = 1; j <= count ; j++)
            if ( A[j] > A[position_of_max] )
                position_of_max = j;
        ItemType temp = A[count];
        A[count] = A[position_of_max];
        A[position_of_max] = temp;
    }
}
```

This piece of code defines a subroutine template. If you remove the first line, "template<class ItemType>", and substitute the word "int" for the word "ItemType" in the rest of the template, you get a subroutine for sorting arrays of ints. (Even though it says "class ItemType", you can actually substitute any type for ItemType, including the primitive types.) If you substitute "string" for "ItemType", you get a subroutine for sorting arrays of strings. This is pretty much what the compiler does with the template. If your program says "sort(list,10)" where list is an array of ints, the compiler uses the template to generate a subroutine for sorting arrays of ints. If you say "sort(cards,10)" where cards is an array of objects of type Card, then the compiler generates a subroutine for sorting arrays of Cards. At least, it tries to. The template uses the ">" operator to compare values. If this operator is defined for values of type Card, then the compiler will successfully use the template to generate a subroutine for sorting Cards. If ">" is not defined for Cards, then the compiler will fail -- but this will happen at compile time, not, as in Smalltalk, at run time where it would make the program crash.

C++ also has templates for making classes. If you write a template for binary trees, you can use it to generate classes for binary trees of ints, binary trees of strings, binary trees of dates, and so on -- all from one template. The most recent version of C++ comes with a large number of pre-written templates called the **Standard Template Library** or STL. The STL is quite complex. Many people would say that its much too complex. But it is also one of the most interesting features of C++.

Generic Programming in Java

Like C++, Java is a strongly typed language. However, generic programming in Java is closer in spirit to Smalltalk than it is to C++. As I've already noted, generic programming in Java is based on the fact that class `Object` is a superclass of every other class. To some extent, this makes Java similar to Smalltalk: A data structure designed to hold `Objects` can hold values belonging to any class. There is no need for templates or any other new language feature to support generic programming.

Of course, primitive type values, such as integers, are not objects in Java and therefore cannot be stored in generic data structures. In fact, there is no way to do generic programming with the primitive data types in Java. The Smalltalk approach doesn't work except for objects, and the C++ approach is not available. Furthermore, generic subroutines are more problematic in Java than they are in either Smalltalk or C++. In Smalltalk, a subroutine can be called with parameter values of any type, and it will work fine as long as all the operations used by the subroutine are supported by the actual parameters. In Java, parameters to a subroutine must be of a specified type, and the subroutine can only use operations that are defined for that type. A subroutine with a parameter of type `Object` can be applied to objects of any type, but the subroutine can only use operations that are defined in class `Object`, and there aren't many of those! For example, there is no comparison operation defined in the `Object` class, so it is not possible to write a completely generic sorting algorithm. We'll see below how Java addresses this problem.

Because of problems like these, some people (including myself) claim that Java does not really support true generic programming. Other people disagree. But whether it's true generic programming or not, that doesn't prevent it from being very useful.

Collections and Maps

Java's generic data structures can be divided into two categories: **collections** and **maps**. A collection is more or less what it sounds like: a collection of objects. A map associates objects in one set with objects in another set in the way that a dictionary associates definitions with words or a phone book associates phone numbers with names. A map is similar to what I called an "association list" in [Section 8.4](#).

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. A set is a collection in which no object can appear more than once.

Note that the terms "collection," "list," "set," and "map" tell you nothing about how the data is stored. A list could be represented as an array, as a linked list, or, for that matter, as a map that associates the elements of the list to the numbers 0, 1, 2, . . . In fact, these terms are represented in Java not by classes but by interfaces. The interfaces `Collection`, `List`, `Set`, and `Map` specify the basic operations on data structures of these types, but do not specify how the data structures are to be represented or how the operations are to be implemented. That will be specified in the classes that implement the interfaces. Even when you use these classes, you might not know what the implementation is unless you go look at the source code. Java's generic data structures are **abstract data types**. They are defined by the operations that can be performed on them, not by the physical layout of the data in the computer.

We will look at list and set classes in [Section 2](#) and map classes in [Section 3](#). But before we do that, we'll look briefly at some of the general operations that are available for all collections.

Generic Algorithms and Iterators

The `Collection` interface includes methods for performing some basic operations on collections 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 any object that implements the `Collection` interface. Here are some of the operations that are defined:

- `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.contains(object)` -- returns a boolean value that is `true` if `object` is in the collection.
- `coll.add(object)` -- adds `object` to the collection. The parameter can be any `Object`. Some collections can contain the value `null`, while others cannot. This method returns a boolean value which tells you whether the operation actually modified the collection. For example, adding an object to a `Set` has no effect if that object was already in the set.
- `coll.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.
- `coll.containsAll(coll2)` -- returns a boolean value that is `true` if every object in `coll2` is also in the `coll`. The parameter can be any `Collection`.
- `coll.addAll(coll2)` -- adds all the objects in the collection `coll2` to `coll`.
- `coll.removeAll(coll2)` -- removes every object from `coll` that also occurs in the collection `coll2`.
- `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`.
- `coll.toArray()` -- returns an array of type `Object[]` that contains all the items in the collection. The return value can be type-cast to another array type, if appropriate. For example, if you know that all the items in `coll` are of type `String`, then `(String[])coll.toArray()` gives you an array of `Strings` containing all the strings in the collection.

Since these methods are part of the `Collection` 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 kinds of `Collection` 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 exception is `UnsupportedOperationException`.

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 you need can be implemented most efficiently. We'll see specific examples of this in the next two sections.

The `Collection` interface 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 infix traversal. Collections can be represented in any of these forms and many others besides. With such a variety of traversal mechanisms, how can we 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 different types of iterators, 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 `Collection` interface 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 an interface named `Iterator`. This interface defines just three methods. If `iter` refers to an `Iterator`, then:

- `iter.next()` -- returns the next item, and advances the iterator. The return value is of type `Object`. 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. 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. 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 that `coll` is of type `Collection`. Then we can say:

```
Iterator iter = coll.iterator();
while ( iter.hasNext() ) {
    Object item = iter.next();
    System.out.println(item);
}
```

The same general form will work for other types of processing. For example, here is a subroutine that will remove all null values from any collection (as long as that collection supports removal of values):

```
void removeNullValues( Collection coll ) {
    Iterator iter = coll.iterator();
    while ( iter.hasNext() ) {
        Object item = iter.next();
        if (item == null)
            iter.remove();
    }
}
```

Collections can hold objects of any type, so the return value of `iter.next()` is `Object`. Now, there's not very much you can do with a general `Object`. In practical situations, a collection is used to hold objects belonging to some more specific class, and objects from the collection are type-cast to that class before they are used. Suppose, for example, that we are working with `Shapes`, where `Shape` is a class that represents geometric figures. Suppose that the `Shape` class includes a `draw()` method for drawing the figure. Then we can write a generic method for drawing all the figures in a collection of `Shapes`:

```
void drawAllShapes( Collection shapeCollection ) {
    // Precondition: Every item in shapeCollection is non-null
    // and belongs to the class Shape.
```

```

        Iterator iter = shapeCollection.iterator();
        while ( iter.hasNext() ) {
            Shape figure = (Shape)iter.next();
            figure.draw();
        }
    }
}

```

The precondition of this method points out that the method will fail if the method contains an item that does not belong to class `Shape`. When that item is encountered, the type-cast "`(Shape)iter.next()`" will cause an exception of type `ClassCastException`. Although it's unfortunate that we can't have a "Collection of Shapes" in Java, rather than a "Collection of Objects", it's not a big problem in practice. You just have to be aware of what type of objects you are storing in your collections.

Equality and Comparison

The discussion of methods in the `Collection` interface had an unspoken assumption: It was assumed that it's known what it means for two objects to be "equal." For example, the methods `coll.contains(object)` and `coll.remove(object)` look for an item in the collection that is equal to `object`. However, equality is not such a simple matter. The obvious technique for testing equality -- using the `==` operator -- does not usually give a reasonable answer when applied to objects. The `==` operator tests whether two objects are identical in the sense that they share the same location in memory. Usually, however, we want to consider two objects to be equal if they represent the same value, which is a very different thing. Two values of type `String` should be considered equal if they contain the same sequence of characters. The question of whether those characters are stored in the same location in memory is irrelevant. Two values of type `Date` should be considered equal if they represent the same time.

The `Object` class defines a boolean-valued method `equals(Object)` for testing whether one object is equal to another. For the purposes of collections, `obj1` and `obj2` are considered to be equal if they are both `null`, or if they are both non-`null` and `obj1.equals(obj2)` is `true`. In the `Object` class, `obj1.equals(obj2)` is defined to be the same as `obj1 == obj2`. However, for most sub-classes of `Object`, this definition is not reasonable, and it should be overridden. The `String` class, for example, overrides `equals()` so that for a `String str`, `str.equals(obj)` if `obj` is also a `String` and `obj` contains the same sequence of characters as `str`.

If you write your own class, you might want to define an `equals()` method in that class to get the correct behavior when objects are tested for equality. For example, a `Card` class that will work correctly when used in collections could be defined as:

```

public class Card { // Class to represent playing cards.
    int suit; // Number from 0 to 3 that codes for the suit --
              // spades, diamonds, clubs or hearts.
    int value; // Number from 1 to 13 that represents the value.
    public boolean equals(Object obj) {
        if (obj == null || !(obj instanceof Card) ) {
            // obj can't be equal to this Card if obj
            // is not a Card, or if it is null.
            return false;
        }
        else {
            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;
    }
}
... // other methods and constructors
}

```

Without the `equals()` method in this class, methods such as `contains()` and `remove()` from the `Collection` interface will not work as expected for values of type `Card`.

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`. This interface defines one method:

```
public int compareTo(Object obj)
```

The value returned by `obj1.compareTo(obj2)` should be zero if and only if the objects are equal (that is, if `obj1.equals(obj2)` is true). It should be negative if and only if `obj1` comes before `obj2`, when the objects are arranged in ascending order. And it should be positive if and only if `obj1` comes after `obj2`. In general, it should be considered an error to call `obj1.compareTo(obj2)` if `obj2` is not of the same type as `obj1`. The `String` class implements the `Comparable` interface and defines `compareTo` in a reasonable way. 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:

```

class FullName implements Comparable {
    // Represents a full name consisting of a first
    // name and a last name.
    String firstName, lastName;
    public boolean equals(Object obj) {
        if (obj == null || !(obj instanceof FullName)) {
            return false;
        }
        else {
            FullName other = (FullName)obj;
            return firstName.equals(other.firstName)
                && lastName.equals(other.lastName);
        }
    }
    public void compareTo(Object obj) {
        Fullname other = (FullName)obj;
        // Will cause an error if obj is not a FullName.
        if ( lastName.compareTo(other.lastName) < 0 ) {
            // If lastName comes before the last name of
            // the other object, then this FullName comes
            // before the other FullName. Return a negative
            // value to indicate this.
            return -1;
        }
        if ( lastName.compareTo(other.lastName) > 0 ) {
            // If lastName comes after the last name of
            // the other object, then this FullName comes
            // after the other FullName. Return a positive
            // value to indicate this.
            return 1;
        }
    }
}

```

```

        else {
            // Last names are the same, so base the comparison
            // on the first names.
            return firstName.compareTo(other.firstName);
        }
    }
    ... // other methods and constructors
}

```

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 `java.util.Comparator`, which defines the method:

```
public int compare(Object obj1, Object obj2)
```

This method compares two objects and returns a value that is negative, or zero, or positive depending on whether `obj1` comes before `obj2`, or is the same as `obj2`, or comes after `obj2`. Comparators are useful for comparing objects that do not implement the `Comparable` interface and for defining several different orderings on the same collection of objects.

In the next two sections, we'll see how `Comparable` and `Comparator` are used in the context of collections and maps.

Wrapper Classes

As noted above, Java's generic programming does not apply to the primitive types. Before leaving this section, we should try to address this problem.

You can't store an integer in a generic data structure designed to hold `Objects`. On the other hand, there is nothing to stop you from making an object that *contains* an integer and putting that object into the data structure. In the simplest case, you could define a class that does *nothing but* contain an integer:

```

public class IntContainer {
    public int value;
}

```

In fact, Java already has a class similar to this one. An object belonging to the class `java.lang.Integer` contains a single `int`. It is called a **wrapper** for that `int`. The `int` value is provided as a parameter to the constructor. For example,

```
Integer val = new Integer(17);
```

creates an `Integer` object that "wraps" the number 17. The `Integer` object can be used in generic data structures and in other situations where an object is required. The `int` value is stored in a private final instance variable of the `Integer` object. If `val` refers to an object of type `Integer`, you can find out what `int` it contains by calling the instance method `val.intValue()`. There is no way to change that value. We say that an `Integer` is an **immutable** object. That is, after it has been constructed, there is no way to change it. (Similarly, an object of type `String` is immutable.)

There are wrapper classes for all the primitive types. All objects belonging to these classes are immutable. The wrapper class for values of type `double` is `java.lang.Double`. The value stored in an object of type `Double` can be retrieved by calling the instance method `doubleValue()`.

The wrapper classes define a number of useful methods. Some of them exist to support generic programming. For example, the wrapper classes all define instance methods `equals(Object)` and `compareTo(Object)` in a reasonable way. Other methods in the wrapper classes are utility functions for

working with the primitive types. For example, we encountered the static methods `Integer.parseInt(String)` and `Double.parseDouble(String)` in [Section 7.4](#). These functions are used to convert strings such as "42" or "2.71828" into the numbers they represent.

[[Next Section](#) | [Chapter Index](#) | [Main Index](#)]

Section 12.2

List and Set Classes

IN THE PREVIOUS SECTION, we looked at the general properties of collection classes in Java. In this section, we look at some specific collection classes and how to use them. These classes can be divided into two categories: lists and sets. A list consists of a sequence of items arranged in a linear order. A list has a definite order, but is not necessarily sorted into ascending order. A set is a collection that has no duplicate entries. The elements of a set might or might not be arranged into some definite order. As with all of Java's collection classes, the items in a list or set are of type `Object`.

The `ListArray` and `LinkedList` Classes

There are two obvious ways to represent a list: as a dynamic array and as a linked list. We've encountered these already in Sections [8.3](#) and [11.2](#). Both of these options are available in generic form as the collection classes `java.util.ListArray` and `java.util.LinkedList`. That is, a `ListArray` represents an ordered sequence of objects stored in an array that will grow in size as new items are added, and a `LinkedList` represents an ordered sequence of objects stored in nodes that are linked together with pointers. Both of these classes implement an interface `java.util.List`, which specifies operations that are available for all lists.

Both list classes support the basic list operations, 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 a linked list, nodes can be added or removed at any position by changing a few pointer values. The `ArrayList` class is more efficient when **random access** to items is required. Random access means accessing the *n*-th item in the list, for any integer *n*. This is trivial for an array, but for a linked list it means starting at the beginning of the list and moving from node to node along the list for *n* steps. 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 `Collection` methods discussed in the [previous section](#), including `size()`, `isEmpty()`, `add(Object)`, `remove(Object)`, and `clear()`. The `add(Object)` method adds the object at the end of the list. The `remove(Object)` method involves first finding the object, which is not very efficient for any list since it involves going through the items in the list from beginning to end until the object is found. The `List` interface adds some methods for accessing list items according to their numerical positions in the list. For an object, `list`, of type `List`, these methods include:

- `list.get(index)` -- returns the `Object` 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 an object `obj` at position number `index` in the list, replacing the object that was there previously. 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`. The number of items in the list increases by one, and items that come after position `index` move up one position to make room for the new item. The value of `index` can be in the range 0 to `list.size()`, inclusive.
- `list.remove(index)` -- removes the object at position number `index`. Items after this position move up one space in the list to fill the hole.
- `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`. If `obj` occurs more than once in the list, the index of the first occurrence is returned.

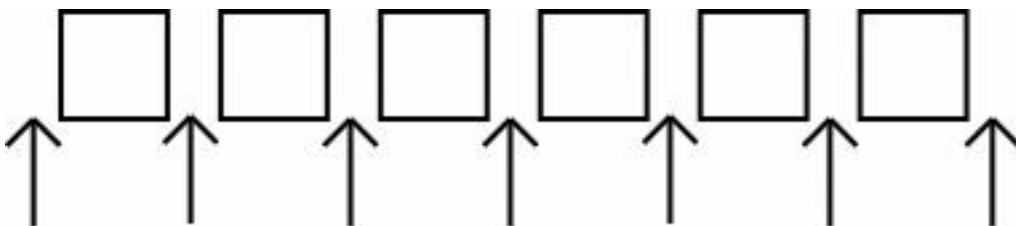
These methods are defined in both the `ArrayList` class and in the `LinkedList` class, although they are only efficient for `ArrayLists`. The `LinkedList` class adds a few additional methods, which are not defined for an `ArrayList`. If `linkedlist` is an object of type `LinkedList`, then

- `linkedlist.getFirst()` -- returns the `Object` that is the first item in the list. The list is not modified.
- `linkedlist.getLast()` -- returns the `Object` 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` as its return value.
- `linkedlist.removeLast()` -- removes the last item from the list, and returns that `Object` as its return value.
- `linkedlist.addFirst(obj)` -- adds the `Object`, `obj`, to the beginning of the list.
- `linkedlist.addLast(obj)` -- adds the `Object`, `obj`, to the end of the list. (This is exactly the same as `linkedlist.add(obj)` and is apparently defined just to keep the naming consistent.)

These methods are apparently defined to make it easy to use a `LinkedList` as if it were a stack or a queue. (See [Section 11.3](#).) For example, we can use a `LinkedList` as a queue by adding items onto one end of the list (using the `addLast()` method) and removing them from the other end (using the `removeFirst()` method).

If `list` is an object of type `List`, then the method `list.iterator()`, defined in the `Collection` interface, 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. The method `list.listIterator()` returns a `ListIterator` for `list`.

A `ListIterator` has the usual `Iterator` methods `hasNext()` and `next()`, but it also has methods `hasPrevious()` and `previous()` that make it possible to move backwards 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 a `ListIterator()`, `iter.next()` moves the iterator one space to the right along the list and returns the item that the iterator passes as it moves. The method `iter.previous()` moves the iterator one space to the left along the list and returns the item that it passes. The method

`iter.remove()` removes an item from the list; the item that is removed is the item that the iterator passed most recently in a call to either `iter.next()` or `iter.previous()`. There is also a method `iter.add(Object)` that adds the specified object to the list at the current position of the iterator. 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 the `LinkedList` class 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 implement efficiently both the `next()` and `previous()` methods of a `ListIterator`. Also, to make the `addLast()` and `getLast()` methods of a `LinkedList` efficient, the `LinkedList` class includes an instance variable that points to the last node in the list.)

As an example of using a `ListIterator`, suppose that we want to maintain a list of items that is always sorted into increasing order. When adding an item to the list, we can use a `ListIterator` to find the position in the list where the item should be added. The idea is to start at the beginning of the list and to move the iterator forward past all the items that are bigger than the item that is being inserted. At that point, the iterator's `add()` method can be used to insert the item at its correct position in the list. In order to say what it means for one item to be "bigger" than another, we assume that the items in the list implement the `Comparable` interface and define the `compareTo()` method. (This interface was discussed in the [previous section](#).) Here is a method that will do this:

```
static void orderedInsert(List list, Comparable newItem) {
    // Precondition:  The items in list are sorted into ascending
    //                order, according to the compareTo method.
    //                newItem.compareTo(item) must be defined for
    //                each item in the list.
    //
    // Postcondition: newItem has been added to the list in its
    //                correct position, so that the list is still
    //                sorted into ascending order.

    ListIterator iter = list.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()) {
        Object 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);
} // end orderedInsert()
```

Since the parameter in this method is of type `List`, it can be applied to both `ArrayLists` and `LinkedLists`, and it will be about equally efficient for both types of lists. You would probably find it easier to write an `orderedInsert` method using array-like indexing with the methods `get(index)` and `add(index, obj)`. However, that method would be horribly inefficient for `LinkedLists` because `get(index)` is so inefficient for such lists. You can find a program that tests this method in the file [ListInsert.java](#).

Sorting

Sorting a list is a fairly common operation, and there should really be a sorting method in the `List` interface. For some reason, there is not, but methods for sorting `Lists` are available as static methods in the class `java.util.Collections`. This class contains a variety of static utility methods for working with collections. The command

```
Collections.sort(list);
```

can be used to sort a list into ascending order. The items in the list must implement the `Comparable` interface. This method will work, for example, for lists of `Strings`. If a `Comparator` is provided as a second argument:

```
Collections.sort(list, comparator);
```

then 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 4](#).

The `Collections` class has at least two other useful methods for modifying lists.

`Collections.shuffle(list)` will rearrange the elements of the list into a random order.

`Collections.reverse(list)` will reverse the order of the elements, so that the last element is moved to the beginning of the list, the next-to-last element to the second position, and so on.

Since an efficient sorting method is provided for `Lists`, there is no need to write one yourself. You might be wondering whether there is an equally convenient method for standard arrays. The answer is yes.

Array-sorting methods are available as static methods in the class `java.util.Arrays`. The command:

```
Arrays.sort(A);
```

will sort an array, `A`, provided either that the base type of `A` is one of the primitive types (except `boolean`) or that `A` is an array of `Objects` that implement the `Comparable` interface. You can also sort part of an array. This is important since arrays are often only "partially filled." The command:

```
Arrays.sort(A, fromIndex, toIndex);
```

sorts the elements `A[fromIndex]`, `A[fromIndex+1]`, ..., `A[toIndex-1]` into ascending order. You can use `Arrays.sort(A, 0, N)` to sort a partially filled array which has elements in the first `N` positions.

Java does not support generic programming for primitive types. In order to implement the command `Arrays.sort(A)`, the `Arrays` class contains eight methods: one method for arrays of `Objects` and one method for each of the primitive types `byte`, `short`, `int`, `long`, `float`, `double`, and `char`.

The TreeSet and HashSet Classes

A set is a collection of Objects in which no object occurs more than once. Objects `obj1` and `obj2` are considered to be the same if `obj1.equals(obj2)` is true, as discussed in the previous section. Sets implement all the general Collection methods, 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`, 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 `Set` interface: `java.util.TreeSet` and `java.util.HashSet`.

In addition to being a `Set`, a `TreeSet` has the property that the elements of the set are arranged into ascending sorted order. An `Iterator` 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 `TreeSet` should implement the `Comparable` interface and that `obj1.compareTo(obj2)` should be defined in a reasonable way for any two objects `obj1` and `obj2` in the set. Alternatively, a `Comparator` can be provided as a parameter to the constructor when the `TreeSet` is created. In that case, the `Comparator` will be used to compare objects that are added to the set.

In the implementation of a `TreeSet`, the elements are stored in something like a binary sort tree. (See [Section 11.4](#).) The actual type of tree that is used is **balanced** in the sense that all the leaves of the tree are at about the same distance from the root of the tree. The number of operations required to find an item in a sorted tree is the same as the distance from the root of the tree to the item. Using a balanced tree ensures that all items are as close to the root as possible. This makes finding an item very efficient. Adding and removing elements are equally efficient.

The fact that a `TreeSet` sorts its elements and removes duplicates makes it very useful in some applications. In [Section 10.3](#), I presented a program, [WordList.java](#), that reads all the words in a file and outputs a list of the words it found. The list is sorted and duplicates have been removed. In that program, I used a linked list to store the words and had to write a subroutine to make sure that the list was sorted and contained no duplicates. By using a `TreeSet` instead of a list, that part of the programming is taken care of automatically. An algorithm for the program, using a `TreeSet`, would be:

```
TreeSet words = new TreeSet();

while there is more data in the input file:
    Let word = the next word from the file.
    words.add(word);

Iterator iter = words.iterator();
while (iter.hasNext()):
    Write iter.next() to the output file.
```

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` (or 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 set = new TreeSet();
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 set = new TreeSet();
set.addAll(coll);
ArrayList list = new ArrayList();
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, `new TreeSet(coll)` creates a `TreeSet` that contains the same elements as the `Collection`, `coll`. This means that we can abbreviate the four lines in the above example to the single command:

```
ArrayList list = new ArrayList( new TreeSet(coll) );
```

This makes a sorted list of the elements of `coll` with no duplicates. A nice example of the power of generic programming. (It seems, by the way, there is no equally easy way to get the list *with* duplicates. To do this, we would need something like a `TreeSet` that allows duplicates. The C++ programming language has such a thing and refers to it as a **multiset**. The Smalltalk language has something similar and calls it a **bag**. Java, for the time being at least, lacks this data type.)

A `HashSet` stores its elements in a **hash table**, a type of data structure that I will discuss in the [next section](#). The operations of finding, adding, and removing elements are implemented very efficiently in hash tables, even more so than for `TreeSets`. The elements of a `HashSet` are not stored in any particular order. An `Iterator` for a `HashSet` will visit its elements in what seems to be a completely arbitrary order, and it's possible for the order to change if a new element is added. Because the elements of a `HashSet` are not ordered, they do not have to implement the `Comparable` interface. 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.

[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Section 12.3

Map Classes

AN ARRAY OF N ELEMENTS can be thought of as a way of associating some item with each of the integers $0, 1, \dots, N-1$. 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 $0, 1, \dots, N-1$, but for arbitrary `Objects`. In fact, some programming languages use the term **associative array** instead of "map" and use the same notation for associative arrays as for regular arrays. In those languages, for example, you might see the notation `A["fred"]` used to indicate the item associated to the string "fred" in the associative array `A`. Java does not use array notation for maps, but the idea is that same: A map is like an array, but the indices for a map are arbitrary 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.

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. If `map` is a variable of type `Map`, then these are some of the methods that are defined for `map`:

- `map.get(key)` -- returns the `Object` that is associated by the map to the `Object` `key`. If the map does not associate any value with `obj`, 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` and `value` can be any objects. 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 any other map, 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 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 `Map` interface: `TreeMap` and `HashMap`. 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 Comparable interface, or that a Comparator must be provided for comparing keys. (The Comparator can be provided as a parameter to the TreeMap constructor.)

A HashMap does not store associations in any particular order, so there are no restrictions on the keys that can be used in a HashMap. 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 use a HashMap.

Let's look at an example. In [Section 8.4](#), 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. Using a Map does have the disadvantage that we are forced to work with Objects instead of Strings. If that is a problem, it's easy to define a phone directory class that uses a Map in its implementation:

```
import java.util.HashMap;

public class PhoneDirectory {

    private HashMap info = new HashMap(); // Stores the data for
                                           // the phone directory.

    public void addEntry(String name, String number) {
        // Record the phone number for a specified name.
        info.put(name, number);
    }

    public String getNumber(String name) {
        // Retrieve the phone number for a specified name.
        // Returns null if there is no number for the name.
        return (String)info.get(name);
    }

} // end class PhoneDirectory
```

In the definition of the getNumber() method, the return value of info.get(name) is type-cast to type String. Since the return type of the get() method is Object, a type-cast is typically necessary before the return value can be used. By "wrapping" the Map in a PhoneDirectory class, we hide this unsightly type-cast in the implementation of the class and provide a more natural interface for the phone directory.

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, then the method:

```
map.keySet()
```

returns the set of all objects that occur as keys for associations in the map. That is, the return value is an object that implements the Set interface. 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 objects that are stored in the map. This "view" of the map implements the `Set` 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:

```
Set keys = map.keySet();           // The set of keys in the map.
Iterator keyIter = keys.iterator();
System.out.println("The map contains the following associations:");
while (keyIter.hasNext()) {
    Object key = keyIter.next();    // Get the next key.
    Object value = map.get(key);    // Get the value for that 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.

The `Map` interface defines two other views. If `map` is a variable of type `Map`, then the method:

```
map.values()
```

returns a `Collection` 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 information in this class 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 element of the entry set is an object belonging to the class `Map.Entry`. (This class is defined as a static nested class, so its full name contains a period. However, it can be used in the same way as any other class to declare variables and do type-casts.) A `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. We could use the entry set of a map to print all the keys and values. 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 look up the value associated with each key:

```
Set entries = map.entrySet();
Iterator entryIter = entries.iterator();
System.out.println("The map contains the following associations:");
while (entryIter.hasNext()) {
    Map.Entry entry = (Map.Entry)entryIter.next();
    Object key = entry.getKey();    // Get the key from the entry.
    Object value = entry.getValue(); // Get the value.
    System.out.println( "    (" + key + ", " + value + ")" );
}
```

Maps are not the only place in Java's generic programming framework where views are used. For example, the `List` interface defines a **sub-list** as a view of a part of a list. The method:

```
List subList(int fromIndex, int toIndex)
```

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 being made to the original list.

Similarly, it is possible to obtain views that represent certain subsets of a sorted set. If `set` is a `TreeSet`, then `set.subSet(fromElement, toElement)` returns a `Set` that contains all the elements of `set` that are between `fromElement` and `toElement` (including `fromElement` and excluding `toElement`). For example, if `words` is a `TreeSet` 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 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 `TreeMap` class 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`, 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 in the obvious way. Suppose, for example, that `blackBook` is a `TreeMap` in which the keys are names and the values are phone numbers. We can print out all the entries from `blackBook` where the name begins with "M" as follows:

```
Map ems = blackBook.subMap("M", "N");
    // This submap contains entries for which the key is greater
    // than or equal to "M" and strictly less than "N".

if (ems.isEmpty())
    System.out.println("No entries beginning with M.");
else {
    Iterator iter = ems.entrySet().iterator();
    // This iterator will traverse the entries in the submap, ems.
    while (iter.hasNext()) {
        // Get the next entry and print its key and value.
        Map.Entry entry = iter.next();
        System.out.println( entry.getKey() + ": " + entry.getValue() );
    }
}
```

Subsets and submaps are probably best thought of as generalized search operations that make it possible to find all the items in a range of values, rather than just to find a single value. Suppose, for example that a database of scheduled events is stored in a `TreeMap` in which the keys are the times of the events, and suppose you want a listing of all events that are scheduled for some time on July 4, 2002. Just make a submap containing all keys in the range from 12:00 AM, July 4, 2002 to 12:00 AM, July 5, 2002, and output all the entries from that submap. This type of search, which is known as a **subrange query** is quite common.

Hash Tables

`HashSets` and `HashMaps` are implemented using a data structure known as a **hash table**. You don't need to understand hash tables to use `HashSets` or `HashMaps`, but any computer programmer should be familiar with hash tables and how they work.

Hash tables are an elegant solution to the search problem. A hash table, like a `HashMap`, stores key/value

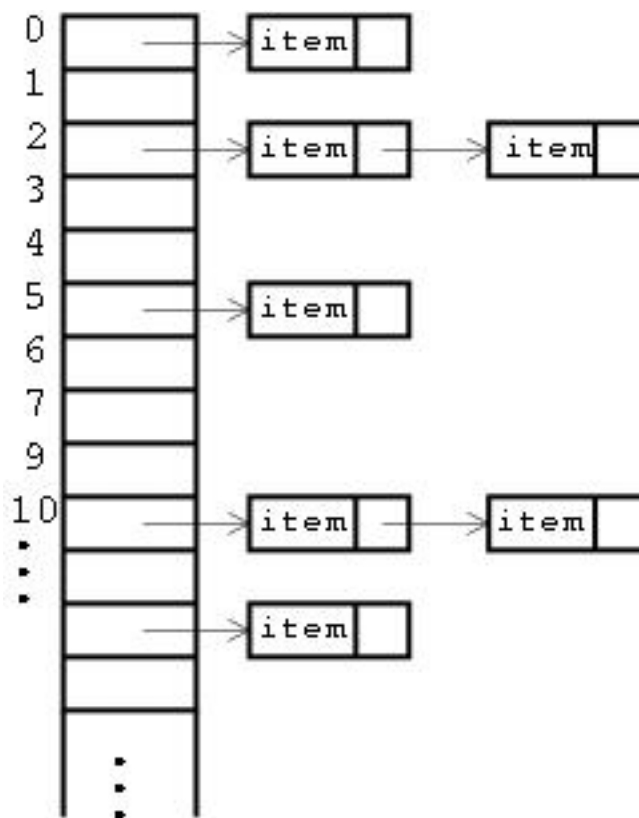
pairs. Given a key, you have to search the table for the corresponding key/value pair. When a hash table is used to implement a set, the values are all null, and the only question is whether or not the key occurs in the set. You still have to search for the key to check whether it is there or not.

In most search algorithms, in order to find the item you are interested in, you have to look through a bunch of other items that don't interest you. To find something in an unsorted list, you have to go 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 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 *N* would be stored in *A*[*N*]. The key takes us directly to the location of the key/value pair. The problem is that there are usually far too many different possible keys for us to be able to use an array with one location for each possible key. For example, if the key can be any value of type `int`, then we would need an array with over four billion locations -- quite a waste of space if we are only going to store, say, a few thousand items! If the key can be a string of any length, then the number of possible keys is infinite, and using an array with one location for each possible key is simply impossible.

Nevertheless, hash tables store their data in an array, and the array index where a key is stored is based on the key. The index is not equal to the key, but it is computed from the key. The array index for a key is called the **hash code** for that key. A function that computes a hash code, given a key, is called a **hash function**. To find a key in a hash table, you just have to compute the hash code of the key and go directly to the array location given by that hash code. If the hash code is 17, look in array location number 17.

Now, since there are fewer array locations than there are possible keys, it's possible that we might try to store two or more keys in the same array location. This is called a **collision**. A collision is not an error. We can't reject a key just because another key happened to have the same hash code. A hash table must be able to handle collisions in some reasonable way. In the type of hash table that is used in Java, each array location actually holds a linked list of key/value pairs (possibly an empty list). When two items have the same hash code, they are in the same linked list. The structure of the hash table looks something like this:



In this diagram, there is one item with hash code 0, no items with hash code 1, two items with hash code 2, and so on. In a properly designed hash table, most of the linked lists are of length zero or one, and the average length of the lists is less than one. Although the hash code of a key doesn't necessarily take you directly to that key, there are probably no more than one or two other items that you have to look through before finding the key you want. For this to work properly, the number of items in the hash table should be somewhat less than the number of locations in the array. In Java's implementation, whenever the number of items exceeds 75% of the array size, the array is replaced by a larger one and all the items in the old array are inserted into the new one.

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 can be computed as $\text{Math.abs}(\text{obj.hashCode()}) \% N$, 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 should have the same hash code. In the `Object` class, both `equals()` and `hashCode()` are based on the address of the memory location where the object is stored. However, as noted in [Section 1](#), many classes redefine the `equals()` method. If a class redefines the `equals()` method, and if objects of that class will be used as keys in hash tables, then the class should also redefine the `hashCode()` method. For example, in the `String` class, the `equals` method is redefined so that two objects of type `String` are considered to be equal if they contain the same sequence of characters. The `hashCode()` method is also redefined in the `String` class, so that the hash code of a string is computed from the characters in that string rather than from its location in memory. For Java's standard classes, you can expect `equals()` and `hashCode()` to be correctly defined. However, you might need to define these methods in classes that you write yourself.

Section 12.4

Programming with Collection Classes

IN THIS SECTION, we finish the chapter and the book by looking at a few programming examples that use the collection and map classes which were discussed in Sections 1, 2, and 3.

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 variables' values. The keys for the symbol table are variable names. The value associated with a key is the value of that variable.

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 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 [Section 11.5](#), we saw how to write a program, [SimpleParser2.java](#), that can evaluate expressions that do not contain variables. The new program, [SimpleParser5.java](#), is based on that older program. I will only talk about the parts that are relevant to the symbol table. Here is an applet that simulates the program:

(Applet "SimpleParser5Console" would be displayed here
if Java were available.)

The program uses a HashMap as the symbol table. A TreeMap could also be used, but since we don't have any reason to access the variables in alphabetical order, we don't need to have the keys stored in sorted

order. The value of a variable is a `double`, but Java's collection and map classes can only hold objects. To get around this restriction, we have to use the wrapper class, `Double`. The variable's value, which is of type `double` is wrapped in an object of type `Double`. That object is stored in the `HashMap`, using the variable's name as the key.

Let `symbolTable` be the `HashMap` that is used as the symbol table. At the beginning of the program, we start with an empty map:

```
symbolTable = new HashMap();
```

To execute a "let" command, the program uses the `put()` method to associate a value with the variable name. Suppose that the name of the variable is given by a `String`, `varName`. The command for setting the value of the variable to `val` would be:

```
symbolTable.put(varName, new Double(val));
```

This associates the object `new Double(val)` with the key, `varName`. In the program [SimpleParser5.java](#), you'll find this in the method named `doLetCommand()`. Just for fun, I decided to pre-define two variables named "pi" and "e" whose values are the usual mathematical constants *pi* 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", new Double(Math.PI));
symbolTable.put("e", new double(Math.E));
```

When a variable is encountered in an expression, the `get()` method is used to retrieve its value. Since this method returns a value of type `Object`, we have to type-cast the return value to `Double`. If the variable has never been given a value, then the `get()` method returns `null`. We have to handle this in some way; I will consider it to be an error:

```
Object symbolTableEntry = symbolTable.get(varName);
if (symbolTableEntry == null) {
    ... // Throw an exception: Undefined variable.
}
Double value = (Double)symbolTableEntry;
double val = value.doubleValue();
```

The last line gets the `double` that is the actual value of the variable from the wrapper object. You will find this code, more or less, in a method named `primaryValue()` in the program [SimpleParser5.java](#).

As you can see, aside from the necessity of using a wrapper class, `Maps` are really quite easy to use.

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 values 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 a 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. We have to wrap each value of type `int` in an object belonging to the wrapper class, `Integer`.

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`. The `TreeSet` corresponding to a term contains `Integers` which give the page numbers of every page on which the term appears.

To make an index, we need to start with an empty `TreeMap`, look through the book, inserting every reference that we want to be in the index into the `TreeMap`, and then print out the data from the `TreeMap`. 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. The `TreeMap` can be created with the command:

```
TreeMap index = new TreeMap();
```

Now, suppose that we find a reference to some term on some page. 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 already have 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, and we should just add the new page reference to the set. Here is a subroutine that does this:

```
void addReference(String term, int pageNum) {
    // Add a page reference to the index.
    TreeSet references; // The set of page references that we
                        // have so far for the term.
    references = (TreeSet)index.get(term); // Type-cast!
    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 firstRef = new TreeSet();
        firstRef.add( new Integer(pageNum) );
        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.
        references.add( new Integer(pageNum) );
    }
}
```

The only other thing we need to do with the index is print it out. We are going to have to print out sets of `Integers`. Let's write a separate subroutine to do that:

```
void printIntegers( Set integers ) {
    // Assume that all the objects in the set are of
    // type Integer. Print the integer values on
    // one line, separated by commas. The commas
    // make this a little tricky.
```

```

        if (integers.isEmpty()) {
            // There is nothing to print if the set is empty.
            return;
        }
        Integer integer; // One of the Integers in the set.
        Iterator iter = integers.iterator(); // For traversing the set.
        integer = (Integer)iter.next(); // First item in the set.
                                           // We know the set is non-empty,
                                           // so this is OK.
        System.out.print(integer.intValue()); // Print the first item.
        while (iter.hasNext()) {
            // Print additional items, if any, with separating commas.
            integer = (Integer)iter.next();
            System.out.print(", " + integer.intValue());
        }
    }
}

```

Finally, we need a routine that can iterate through all the terms in the map and print each term along with its list of page references. There are no iterators for maps. To iterate through a map, we need to use one of the Set views of the map. Since we want to print both the keys and the values, it's most efficient to use the entry set of the map. Here's the subroutine:

```

void printIndex() {
    // Print each entry in the index.

    Set entries = index.entrySet();
        // The index viewed as a set of entries, where each
        // entry has a key and a value. The objects in
        // this set are of type Map.Entry.

    Iterator iter = entries.iterator();

    while (iter.hasNext()) {
        // Get the next entry from the entry set and print
        // the term and list of page references that
        // it contains.
        Map.Entry entry = (Map.Entry)iter.next();
        String term = (String)entry.getKey();
        Set pages = (Set)entry.getValue();
        System.out.print(term + " ");
        printIntegers(pages);
        System.out.println();
    }
}

```

This is not a lot of code, considering the complexity of the operations. The only really tricky part is the constant need for type-casting and the need to use wrapper objects for primitive types. Both of these are necessary because of the nature of generic programming in Java, that is, the fact that generic data structures hold values of type `Object`.

I have not written a complete indexing program, but the subroutines presented here are used in a related context in [Exercise 12.5](#).

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 a `Comparator` is an object that implements the `Comparator` interface and defines the method:

```
public int compare(Object obj1, Object obj2)
```

This method can be used to compare two objects. It should return an integer that is positive, zero, or negative, depending on whether `obj1` is less than, equal to, or greater than `obj2`. We need a `Comparator` that will compare two `Strings` based on alphabetical order. The easiest (although not most efficient) way to do this is to convert the `Strings` to lower case and use the default comparison on the lower case `Strings`. The following class defines such a comparator:

```
class AlphabeticalOrder implements Comparator {
    // Represents a Comparator that can be used for
    // comparing Strings according to alphabetical
    // order. It is an error to apply this
    // Comparator to objects that are non-strings.
    public int compare(Object obj1, Object obj2) {
        String str1 = (String)obj1; // Type-cast objects to Strings.
        String str2 = (String)obj2;
        str1 = str1.toLowerCase(); // Convert to lower case.
        str2 = str2.toLowerCase();
        return str1.compareTo(str2); // Compare lower-case Strings.
    }
}
```

To solve our indexing problem, we just need to tell our index to use an object of type `AlphabeticalOrder` for comparing keys. This is done by providing the `Comparator` object as a parameter to the constructor. We just have to create the index with the command:

```
TreeMap index = new TreeMap( new AlphabeticalOrder() );
```

This does work, but I've concealed one technicality. Suppose, for example, that the 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 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. 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. This is probably the correct behavior in this example. If not, some other technique must be used to sort the terms into alphabetical order.

Word Counting

The final example also deals with storing information about words. In [Section 10.3](#), we looked at [WordList.java](#), a program that makes a list of all the words in a file. In [Section 2](#), we did the same thing using generic data structures. Now, we extend the problem so that instead of just listing the words, the program also counts the number of times each word occurs in the file. The output consists of two lists, one sorted alphabetically and one sorted according to the number of occurrences, with the most common words at the top and the least common at the bottom. The same problem was assigned in [Exercise 10.1](#) and solved without using generic data structures.

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 storage 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:

```
static class WordData {
    // Represents the data we need about a word:  the word and
    // the number of times it has been encountered.
    String word;
    int count;
    WordData(String w) {
        // Constructor for creating a WordData object when
        // we encounter a new word.
        word = w;
        count = 1; // The initial value of count is 1.
    }
} // end class WordData
```

The program has to store all the `WordData` objects in some sort of data structure. We want to be able to add new words efficiently. Given a word, we need to check whether a `WordData` object already exists for that word, and if it does, we need to find that object so that we can increment its counter. A `Map` can be used to implement these operations. Given a word, we want to look up a `WordData` object in the `Map`. This means that the word is the *key*, and the `WordData` object is the *value*. (It might seem strange that the key is also one of the instance variables in the value object, but in fact this is probably the most common situation: The value object contains all the information about some entity, and the key is one of those pieces of information.) 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.

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, where `words` is a variable that refers to the `TreeMap`. 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 it's the `WordData` object for this word, and the program only has to increment the counter in that object. Here is the subroutine that is used to read the words from the file:

```
static void readWords(Reader inStream, Map words) {
    // Read all words from inStream, and store data about them in words.

    try {
        while (true) {
            while (! inStream.eof() && ! Character.isLetter(inStream.peek()))
                inStream.getAnyChar(); // Skip past non-letters.
```

```

        if (inStream.eof())
            break; // Exit because there is no more data to read.
        String word = inStream.getAlpha(); // Read one word from stream.
        word = word.toLowerCase();
        WordData data = (WordData)words.get(word);
        // Check whether the word is already in the Map. If not,
        // the value of data will be null. If it is not null, then
        // it is a WordData object containing the word and the
        // number of times we have encountered it so far.
        if (data == null) {
            // We have not encountered word before. Add it to
            // the map. The initial frequency count is
            // automatically set to 1 by the WordData constructor.
            words.put(word, new WordData(word) );
        }
        else {
            // The word has already been encountered, and data is
            // the WordData object that holds data about the word.
            // Add 1 to the frequency count in the WordData object.
            data.count = data.count + 1;
        }
    }
}
catch (TextReader.Error e) {
    System.out.println("An error occurred while reading the data.");
    System.out.println(e.toString());
    System.exit(1);
}
} // end readWords()

```

After reading the words and printing them out in alphabetical order, the program has to sort the words by frequency count and print them again. To do the sorting using a generic algorithm, I defined a **Comparator class** for comparing two word objects according to their frequency counts:

```

static class CountCompare implements Comparator {
    // A comparator for comparing objects of type WordData
    // according to their counts. This is used for
    // sorting the list of words by frequency.
    public int compare(Object obj1, Object obj2) {
        WordData data1 = (WordData)obj1;
        WordData data2 = (WordData)obj2;
        return data2.count - data1.count;
        // The return value is positive if data2.count > data1.count.
        // I.E., data1 comes after data2 in the ordering if there
        // were more occurrences of data2.word than of data1.word.
        // The words are sorted according to decreasing counts.
    }
} // end class CountCompare

```

Given this class, we can sort the **WordData** objects according to frequency by copying them into a list and using the generic method `Collections.sort(List, Comparator)`. The **WordData** objects are the values in the map, `words`, and `words.values()` is a **Collection** that contains all these values. The constructor for the **ArrayList** class lets you specify a collection to be copied into the list when it is created. So, we can create a **List** containing the word data and sort that list according to frequency count using the commands:

```

ArrayList wordsByCount = new ArrayList( words.values() );
Collections.sort( wordsByCount, new CountCompare() );

```

You should notice that these two lines replace a lot of code! It requires some practice to think in terms of generic data structures and algorithms, but the payoff is significant in terms of saved time and effort.

The only remaining problem is to print the data to the output file. 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 an `Iterator` for the value collection, `words.values()`, to access the words in alphabetical order. Similarly, the words are in the `ArrayList wordsByCount` in the correct order according to frequency count, so we could use an `Iterator` for the `ArrayList` to access and print the data according to frequency count. Since we have to do essentially the same thing twice, we might as well write a subroutine to do it:

```

static void printWords(PrintWriter outStream, Collection wordData) {
    // wordData must contain objects of type WordData. The words
    // and counts in these objects are written to the output stream.
    Iterator iter = wordData.iterator();
    while (iter.hasNext()) {
        WordData data = (WordData)iter.next();
        outStream.println("    " + data.word + " (" + data.count + ")");
    }
} // end printWords()

```

This is a *generic* subroutine. Since its second parameter is of type `Collection`, it can be applied to the collection `words.values()` as well as to the collection `wordsByCount`. This is the last piece needed for the program. You can find the complete program in the file [WordCount.java](#).

With this section, we reach the end of *Introduction to Programming Using Java*. It has been a long journey, but I hope a worthwhile one and one that has left you prepared to move on to more advanced study of Java, programming, and computer science in general. Good luck and have fun!

End of Chapter 12

[[Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

Programming Exercises For Chapter 12

THIS PAGE CONTAINS programming exercises based on material from [Chapter 12](#) of this [on-line Java textbook](#). Each exercise has a link to a discussion of one possible solution of that exercise.

Exercise 12.1: In [Section 12.2](#), I mentioned that a `LinkedList` can be used as a queue by using the `addLast()` and `removeFirst()` methods to enqueue and dequeue items. But, if we are going to work with queues, it's better to have a `Queue` class. The data for the queue could still be represented as a `LinkedList`, but the `LinkedList` object would be hidden as a private instance variable in the `Queue` object. Use this idea to write a generic `Queue` class for representing queues of `Objects`. Also write a generic `Stack` class that uses either a `LinkedList` or an `ArrayList` to store its data. (Stacks and queues were introduced in [Section 11.3](#).)

[See the solution!](#)

Exercise 12.2: 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: The set `A.addAll(B)` is the *union* of A and B; `A.retainAll(B)` is the *intersection* of A and B; and `A.removeAll(B)` is the *difference* of A and B. (These operations change the contents of the set A, while the mathematical operations create a new set without changing A, but that difference is not relevant to this exercise.)

For this exercise, you should write a program that can be used as a "set calculator" for simple operations on sets of non-negative integers. (Negative integers are not allowed.) A set of such integers will be represented as a list of integers, separated by commas and, optionally, spaces and enclosed in square brackets. For example: `[1,2,3]` or `[17, 42, 9, 53,108]`. The characters `+`, `*`, and `-` will be used for the union, intersection, and difference operations. The user of the program will type in lines of input containing two sets, separated by an operator. The program should perform the operation and print the resulting set. Here are some examples:

Input	Output
-----	-----
<code>[1, 2, 3] + [3, 5, 7]</code>	<code>[1, 2, 3, 5, 7]</code>
<code>[10,9,8,7] * [2,4,6,8]</code>	<code>[8]</code>
<code>[5, 10, 15, 20] - [0, 10, 20]</code>	<code>[5, 15]</code>

To represent sets of non-negative integers, use `TreeSets` containing objects belonging to the wrapper class `Integer`. Read the user's input, create two `TreeSets`, and use the appropriate `TreeSet` method to perform the requested operation on the two sets. Your program should be able to read and process any number of lines of input. If a line contains a syntax error, your program should not crash. It should report the error and move on to the next line of input. (Note: To print out a `Set`, A, of `Integers`, you can just say `System.out.println(A)`. I've chosen the syntax for sets to be the same as that used by the system for outputting a set.)

[See the solution!](#)

Exercise 12.3: 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, you are a computer science student.

Write an implementation of hash tables from scratch. Define the following methods: `get(key)`, `put(key, value)`, `remove(key)`, `containsKey(key)`, and `size()`. Do not use *any* of Java's generic data structures. Assume that both keys and values are of type `Object`, just as for `HashMaps`. Every `Object` has a hash code, so at least you don't have to define your own hash functions. Also, you do *not* have to worry about increasing the size of the table when it becomes too full.

You should also write a short program to test your solution.

[See the solution!](#)

Exercise 12.4: A **predicate** is a boolean-valued function with one parameter. Some languages use predicates in generic programming. Java doesn't, but this exercise looks at how predicates might work in Java.

In Java, we could use "predicate objects" by defining an interface:

```
public interface Predicate {
    public boolean test(Object obj);
}
```

The idea is that an object that implements this interface knows how to "test" objects in some way. Define a class `Predicates` that contains the following generic methods for working with predicate objects:

```
public static void remove(Collection coll, Predicate pred)
    // Remove every object, obj, from coll for which
    // pred.test(obj) is true.

public static void retain(Collection coll, Predicate 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 List collect(Collection coll, Predicate pred)
    // Return a List that contains all the objects, obj,
    // from the collection, coll, such that pred.test(obj)
    // is true.

public static int find(ArrayList list, Predicate pred)
    // Return the index of the first item in list
    // for which the predicate is true, if any.
    // If there is no such item, return -1.
```

(In C++, methods similar to these are included as a standard part of the generic programming framework.)

[See the solution!](#)

Exercise 12.5: One of the examples in [Section 12.4](#) 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 Section 12.4 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. The names of the input file and output file should be specified as command line arguments when the program is run. You can use the indexing subroutines from Section 12.4, modified to write the data to a file instead of to `System.out`. You will also need to make the subroutines `static`. If you need some help with using files and command-line arguments, you can find an example in the sample program [WordCount.java](#), which was also discussed in Section 12.4.

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. Your program will need to keep track of the line number. The end of each line in the file is marked by the newline character, `'\n'`. Every time you encounter this character, add one to the line number. One warning: The method `in.eof()`, which is defined in the `TextReader`, skips over end-of-lines. Since you don't want to skip end-of-line characters, you should not use `in.eof()` -- at least, you should not use it in the same way that it is used in the program `WordCount.java`. The function `in.peek()` from the `TextReader` class returns the nul character `'\0'` at the end of the file. Use this function instead of `in.eof()` to test for end-of-file.

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 12.6: The sample program [SimpleParser5.java](#) from [Section 12.4](#) can handle expressions that contain variables, numbers, operators, and parentheses. Extend this 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.

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 objects belonging to the following class:

```
class StandardFunction {
    // An object of this class represents
    // one of the standard functions.

    static final int SIN = 0, COS = 1,      // Code numbers for each
        TAN = 2, ABS = 3,                  // of the functions.
        Sqrt = 4, LOG = 5;

    int functionCode; // Tells which function this is.
                    // The value is one of the above codes.

    StandardFunction(int code) {
        // Constructor creates the standard function specified
        // by the given code, which should be one of the
        // above code numbers.
        functionCode = code;
    }

    double evaluate(double x) {
        // Finds the value of this function for the
        // specified parameter value, x.
        switch(functionCode) {
            case SIN:
                return Math.sin(x);
```



```

        case COS:
            return Math.cos(x);
        case TAN:
            return Math.tan(x);
        case ABS:
            return Math.abs(x);
        case SQRT:
            return Math.sqrt(x);
        default:
            return Math.log(x);
    }
}

} // end class StandardFunction

```

Add a symbol to the symbol table to represent each function. The key is the name of the function and the value is an object of type `StandardFunction` that represents the function. For example:

```
symbolTable.put("sin", new StandardFunction(StandardFunction.SIN));
```

In 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 value 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!](#)

[[Chapter Index](#) | [Main Index](#)]

Quiz Questions For Chapter 12

THIS PAGE CONTAINS A SAMPLE quiz on material from [Chapter 12](#) of this [on-line Java textbook](#). You should be able to answer these questions after studying that chapter. Sample answers to all the quiz questions can be found [here](#).

Question 1: What is meant by *generic programming* and what is the alternative?

Question 2: Java does not support generic programming with the primitive types. Why not? What is it about generic programming in Java that prevents it from working with primitive types such as `int` and `double`.

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` and that every object in the collection belongs to the wrapper class `Integer`. Write a code segment that will compute the sum of all the integer values in the collection.

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 `Comparable` interface. 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) { // Convenience constructor.
        month = m;
        day = d;
        year = y;
    }
}
```

Question 10: Suppose that `syllabus` is a variable of type `TreeMap`, the keys in the map are objects belonging to the `Date` class from the previous problem, and the values are of type `String`. Write a code segment that will write out the value string for every key that is in the month of September, 2002.

[[Answers](#) | [Chapter Index](#) | [Main Index](#)]

Appendix 1:

Other Features of Java

THIS TEXTBOOK does not claim to cover all the features of the Java programming language, or even to give comprehensive coverage to the features that it does cover. The primary purpose of the book is to explain programming, not Java. Nevertheless, it can serve as a good starting point for learning Java. This appendix briefly surveys some of the features of Java that were *not covered in the book*. It will acquaint you with some of the terms you might hear when people discuss Java, and it will point you towards some of the things you might want to learn more about as you continue your study of Java programming.

JAR Files and Resources

Each programming example in this book uses just a few class files, at most. A large Java project might use hundreds. People might hesitate to welcome a program that comes in hundreds of small files onto their hard drives. Fortunately, Java makes it possible to combine a collection of files into a single **Java archive file**, or "JAR" file. If all the class files needed to run a Java program are placed into a JAR file, then only that one file will be needed. Many Java programming environments can be configured to make JAR files when they compile a program. The command-line programming environment, in which the "javac" command is used for compilation, also has a "jar" command for making JAR files. For example, the following command makes a JAR file named "myprog.jar" and copies all the class files in the current directory into that JAR file:

```
jar cfv myprog.jar *.class
```

The "cfv" means "Create a jar archive as a File and be Verbose about telling me what you are doing." The "*.class" matches all files that end with ".class". The contents of a JAR file are compressed, by default, so the JAR file actually takes up less space than the files it contains.

A JAR file can be used for an applet. It just has to be specified in the <applet> tag:

```
<applet archive="myprog.jar" code="MyApplet.class"
width=200 height=100>
</applet>
```

A JAR file can be used for a stand-alone application by specifying it as part of the "classpath":

```
java -classpath myprog.jar MyApplication
```

In addition to class files, a JAR file can contain images, sounds, and other resource files needed by a program. It's fairly easy to load such resources into a program. An image resource, for example can be loaded in almost the same way as an independent image file, using the `getImage()` method of an applet or a `Toolkit`. The location of the resource just has to be specified differently. For example, if an applet class named `MyApplet` is loaded from a JAR file, and that file contains an image file named "icon.gif", then the following command will load the image:

```
Image icon = getImage( MyApplet.class.getResource("icon.gif") );
```

The `getResource()` method is used to locate a resource. It returns a URL that specifies the location of the resource. (The `getResource()` method in this example is an instance method in an object, `MyApplet.class`, that represents the class to which the applet belongs. Objects that represent classes are another feature of Java that I haven't mentioned before. The idea is that the system will look for the image file in the same places it looked for the class and will find it in the same JAR file.)

Graphics2D

All drawing in Java is done through an object of type `Graphics`. The `Graphics` class provides basic commands for drawing shapes and text and for selecting a drawing color. These commands are adequate in many cases, but they fall far short of what's needed in a serious computer graphics program. Java has another class, `Graphics2D`, that provides a larger and more serious set of drawing operations. `Graphics2D` is a sub-class of `Graphics`, so all the old routines from the `Graphics` class are also available in a `Graphics2D`.

The `paintComponent()` method of a `JComponent` gives you a graphics context of type `Graphics` that you can use for drawing on the component. In fact, the graphics context actually belongs to the sub-class `Graphics2D` (in Java version 1.2 and later), and can be type-cast to gain access to the advanced `Graphics2D` drawing methods:

```
public void paintComponent(Graphics g) {
    Graphics2D g2;
    g2 = (Graphics2D)g;
    .
    . // Draw on the component using g2.
    .
}
```

Drawing in `Graphics2D` is based on shapes, which are objects that implement an interface named `Shape`. **Shape classes include** `Line2D`, `Rectangle2D`, `Ellipse2D`, `Arc2D`, and `CubicCurve2D`, among others. `CubicCurve2D` can be used to draw Bezier Curves, which are used in many graphics programs. `Graphics2D` has commands `draw(Shape)` and `fill(Shape)` for drawing the outline of a shape and for filling its interior. Advanced capabilities include: lines that are more than one pixel thick, dotted and dashed lines, filling a shape with a texture (this is, with a repeated image), filling a shape with a gradient, and drawing translucent objects that will blend with their background.

In the `Graphics` class, coordinates are specified as integers and are based on pixels. The shapes that are used with `Graphics2D` use real numbers for coordinates, and they are not necessarily bound to pixels. In fact, you can change the coordinate system and use any coordinates that are convenient to your application. In computer graphics terms, you can apply a "transformation" to the coordinate system. The transformation can be any combination of translation, scaling, and rotation.

Javadoc

A program that is well-documented is much more valuable than the same program without the documentation. Java comes with a tool called **javadoc** that can make it easier to produce the documentation in a readable and organized format. Javadoc is especially useful for documenting classes and packages of classes that are meant to be used by other programmers. A programmer who wants to use pre-written classes shouldn't need to search through the source code to find out how to use them. If the documentation in the source code is in the correct format, javadoc can separate out the documentation and make it into a set of web pages. The web pages are automatically formatted and linked into an easily browseable Web site. Sun Microsystem's on-line documentation for the standard Java API was produced using javadoc.

Javadoc is actually very easy to use. In a source code file, javadoc documentation looks like a regular multi-line comment, except that it begins with `/**` instead of with `/*`. Each such comment is associated with some class, member variable, or method. The documentation for each item must be placed in a comment that *precedes* the item. (This is how javadoc knows which item the comment is for.) The comments can include certain special notations. For example, the notation `@return` is used to begin the description of the return value of a function. And `@param <parameter-name>` marks the beginning of the description of a parameter of a method. For example, here is a short utility method with a javadoc

comment:

```
/**
 * Return the real number represented by the string s,
 * or return Double.NaN if s does not represent a legal
 * real number.
 *
 * @param s String to interpret as real number.
 * @return the real number represented by s.
 */
public static double stringToReal(String s) {
    try {
        return Double.parseDouble(s);
    }
    catch (NumberFormatException e) {
        return Double.NaN;
    }
}
```

Sun's Java Software Development Kit includes `javadoc` as a program that can be used on the command line. This program takes one or more Java source code files, extracts the javadoc comments from them, and prepares Web pages containing the documentation. Integrated development environments for Java typically include a menu command for generating javadoc documentation.

Internationalization

If the World-Wide Web -- and information technology in general -- is to be a truly global phenomenon, it shouldn't be tied to one country's language or customs. An **internationalized** computer program or applet is one that can adapt itself to the locale where it is being run. A **locale** in Java is specified as a language together with a country. These, in turn, are designated by standard two-letter codes, such as "en" for English, "es" for Spanish, "US" for the United States, "ES" for Spain, and "MX" for Mexico. The locale determines not only the language that is used but also details such as the output format of dates and numbers. A Java virtual machine has a default locale built into it. If it's running in the United States, the default locale is probably `en_US`. In Mexico, it would be `es_MX`.

The classes `java.text.DateFormat` and `java.text.NumberFormat` make it possible to display dates and numbers in a form that is appropriate for the default locale (or some other locale if you want). For example, the commands

```
DateFormat df = DateFormat.getDateInstance();
String now = df.format( new Date() );
System.out.println(now);
```

print the current date and time (as returned by "new Date()"), formatted according to the conventions of the default locale. The output will look different in different countries.

Any text that is displayed to the user by a program -- labels on buttons, commands in menus, messages in dialog boxes, and so on -- should be in the language that is appropriate for the locale where the program is being run. Java makes it possible to put all the strings used by a program into a **resource bundle** and to use a different resource bundle for each locale. The actual strings do not appear in the program itself. They are in separate files. One file might contain the strings in English; another file, the same strings translated into Spanish; and a third file, the same strings in Japanese. The program can then be run in English, Spanish, and Japanese locales, and it will use a different language in each local. To make the program run correctly in a French locale, it's only necessary to create a new resource bundle, with the strings translated into French. It's not necessary to modify the program in any way. The class `java.util.ResourceBundle` represents a set of strings to be used by a program, and makes it possible to load the set of strings that is

most appropriate for the default locale. When the program wants to display a string, it gets the string from the resource bundle. The string will be in the appropriate language for the default locale, assuming that a resource bundle for that language is available.

Pluggable Look-And-Feel

Macintosh programs don't look quite the same or work quite the same as Windows programs. Linux and UNIX have several different GUI styles, but none of them look or work exactly like Macintosh or Windows. Java programs are supposed to work on any computing platform. Ideally, they should have the right look-and-feel for the platform where they are running. In Swing, this is made possible by **pluggable look-and-feel**, or PLAF. The look-and-feel of a Swing interface can be managed using the class `javax.swing.UIManager`, which contains static methods for setting the default look-and-feel of newly created `JComponents` and for determining what look-and-feels are installed on the computer where the program is running. To change the look-and-feel of a `JComponent` that already exists, you can call its `updateUI()` method (after calling `UIManager.setLookAndFeel()`). To change the look-and-feel of a component and all the components that it contains, you can use `SwingUtilities.updateComponentTreeUI(comp)`, where `comp` is the component.

Java has a cross-platform look-and-feel called "Metal" that will look about the same on all platforms. Other look-and-feels might be available, depending on the platform and the Java virtual machine that is being used. On MacOS X, for example, the default look-and-feel for Java is similar to the Mac's "Aqua" interface.

PLAF is supported by the basic architecture of Swing components. Swing uses a **Model-View-Controller** architecture in which the model (that is, the data) for a component is separate from the view (that is, the on-screen visual representation of the data). The data for a `JButton` is stored in an object of type `ButtonModel`. The data for a `JTextComponent` is stored in an object of type `Document`. When the look-and-feel is changed, the model is not affected, but the view can change. Most of the time, you don't need to be aware of the distinction between components and their models, but in some cases you need access to the model. For example, if you want to install a listener to respond to each change that is made to the contents of a `JTextComponent`, you have to register a `DocumentListener` with the text component's model: `textInput.getDocument().addDocumentListener(listener)`.

JavaBeans

A JavaBean is a component that can be combined with other components to make a complete program. JavaBeans can be assembled into a program using a **visual development environment**, which allows a programmer to add beans to a program, configure them, and set up interactions between them by dragging icons, using menus, clicking buttons, and so on. Sophisticated programs can be assembled with little or no programming. If you use an integrated development environment for Java programming, there's a good chance that it has some support for visual programming.

JavaBeans are objects, not classes. Many JavaBeans are GUI components, but this is not a requirement and a JavaBean might have no visual representation at all. Objects belonging to Java's standard GUI component classes are JavaBeans and can be used in visual development environments.

JavaBeans can be defined by any class that follows a few rules. The class should have a default constructor (one with no parameters). This allows a visual development environment to create a new bean in a default state without providing any information to the constructor. To be useful, a bean should have one or more **properties**, which are just values associated with the bean. Beans are configured by setting the values of their properties. A visual development environment recognizes a property by the fact that there are "get" and "set" methods for the bean. For example, if a bean has methods:

```
String getTitle();
```



```
void setTitle(String title);
```

then the bean has a property named "Title" of type `String`. You don't have to do anything else to define a property; just provide the "get" and "set" methods. Beans can, optionally, use `PropertyChangeEvents` to communicate. When a property of a bean is changed, it can emit an event. Other beans can register to listen for these events, so that they can respond when the property changes. Of course, beans might also generate other types of events, such as `ActionEvents`. A visual development environment should make it possible to route these events to other beans or possibly to write some code to respond to the events.

JavaBeans exist to enable the production of reusable objects and to promote the development of an "object economy" in which such objects are widely distributed and readily available.

Distributed Computing

Java is a language that was designed from the beginning to work in a networked computing environment. Applets can be downloaded over a network, and basic network communication is supported by the `java.net` package. But this is just the beginning. Java has built-in support for **distributed computing**. In distributed computing, a program uses more than one computer. Different parts of the program run on different computers and communicate over a network. The program has access to much larger computing resources than are available on any one computer.

In Java, this is made possible by allowing a method that is running on one computer to call a method in an object that is on another computer. The parameters for the method are transmitted across the network, and the return value is sent back after the method has completed.

Java has support for two types of distributed object computing. **RMI** (Remote Method Invocation) is used for communication between two Java objects running on different computers. Java also supports **CORBA** (Common Object Request Broker Architecture), a standard that allows communication between objects written in different programming languages.

Servlets and JSP

Java is most visible on the client side of the Web, in the form of applets running in Web browsers. However, Java is also very useful on the server side. A **servlet** is a Java program that is meant to be executed by a Web server. This is similar to the way that applets are executed by a Web browser. If a Web server is capable of executing servlets, then its capabilities can be extended indefinitely by writing new servlets for it. Java is certainly not the only programming language used in this way, but it is an attractive choice because of its security, network-awareness, and large collection of APIs.

Servlets are often used to generate Web pages. Many Web pages are **static** -- they are simple, unchanging HTML files. When a server receives a request for such a page from a Web browser, all it has to do is send the HTML file to the browser. However, there are also **dynamic** Web pages. A dynamic web page is generated on demand each time the page is requested. The content of the page can be different each time it is requested. This can happen, for example, because the content depends on data that was typed into a Web form by the user or on information from a database. To serve up a dynamic page, the server has to run a program. Servlets can be used in this way. The servlet decides what should be on the page and creates the HTML code for displaying that content. The server then sends this HTML code to the Web browser that requested the page.

The task of writing dynamic Web pages can be simplified by using **Java Server Pages** (JSP) instead of servlets. A Java server page is an HTML file with some Java code embedded in it. When a Web server receives a request for the page, the Java code is executed to generate the dynamic part of the page. JSP is used on many Web sites. You can recognize a JSP page because the file name for the page will end with

".jsp".

Servlets and JSP are not part of the standard edition of Java, J2SE, but they are included in the enterprise edition, J2EE.

More Features

Java has APIs (Application Programming Interfaces) for many more features, and the list seems to be growing all the time. It's unlikely that any one person can master them all. What can be mastered are the principles and techniques on which they are all built. After that, it's just a matter of poking around in the documentation . . . Here are a few more of the things you might run into:

- **Multimedia.** The packages `java.awt.image` and `javax.swing.sound` contain classes for manipulating images and sounds.
- **Drag-and-Drop.** Drag-and-drop refers to dragging an item that is to be processed and dropping it onto the item that you want to process it. An example is dragging a file and dropping it into the trash. Drag-and-Drop support in Java is provided in the package `java.awt.dnd`.
- **Accessibility.** Not everyone can see a computer screen, hear sounds, use a mouse, and type on a keyboard. A typical user interface is not accessible to these people. Java has an infrastructure that can be used to make programs accessible. It is defined in the package `javax.accessibility`.
- **Security.** The package `java.security` can be used for secure, encrypted network communication.
- **Database.** JDBC (Java DataBase Connectivity) refers to set of classes that is used to connect to databases and retrieve information from them. The basic classes are defined in the package `java.sql` (but to use them, you also need a "driver" for the specific type of database that you want to connect to).
- **XML.** XML is a data representation format that is similar to HTML. Like HTML, it can be used to describe documents. But it is also used to represent arbitrary structured data. With the release of Java Version 1.4, XML is a standard part of Java. Currently, XML is probably generating more excitement and hype than any other single computing technology.

[[Main Index](#)]

Appendix 2:

Some Notes on Java Programming Environments

ANYONE WHO IS LEARNING to program has to choose a programming environment that makes it possible to create and to run programs. Programming environments can be divided into two very different types: **integrated development environments** and **command-line environments**. All programming environments for Java require some text editing capability, a Java compiler, and a way to run applets and stand-alone applications. An integrated development environment, or IDE, is a graphical user interface program that integrates all these aspects of programming and probably others (such as a debugger, a visual interface builder, and project management). A command-line environment is just a collection of commands that can be typed in to edit files, compile source code, and run programs.

I have programmed using both IDEs and command-line environments, and I have taught programming using both types of environments. Based on my experience, I recommend a command line environment for beginning programmers. IDEs can simplify the management of large numbers of files in a complex project, but they are themselves complex programs that add another level of complications to the already difficult task of learning the fundamentals of programming. Certainly, a serious programmer should have some experience with IDEs, but I think that it's an experience that can be picked up later. This is, of course, just my opinion.

In the rest of this appendix, I'll make a few comments on programming environments. No matter which type of environment you prefer, there is no need to pay for it, so I'll limit my comments to software that is available at no charge. Please note that I am not an expert on Java programming environments. I am including this appendix because people occasionally write to me for help or advice on the matter. In general, however, I *cannot* answer questions about specific programming environments.

The Basics from Sun (and Apple)

Java was developed at Sun Microsystems, Inc., and the primary source for information about Java is Sun's Java Web site, <http://java.sun.com/>. At this site, you can read documentation on-line and you can download documentation and software. You should find some obvious links on the main page. (As of July 1, 2002, they are labeled "Download Now," and a page with various downloads can be found at <http://java.sun.com/j2se/1.4/download.html>.)

The documentation includes the Java API reference and the Java tutorial. These are not really directed at beginning programmers, but you will need them if you are going to be serious about Java programming.

As I write this, the current version of Java on the Sun site is version 1.4. It is available for the Windows, Linux, and Solaris operating systems. You want to download the "J2SE 1.4 SDK." This is the "Java 2 Platform Standard Edition Version 1.4 Software Development Kit." This package includes a Java compiler, a Java virtual machine that can be used to run Java programs, and all the standard Java packages. You want the "SDK", not the "JRE". The JRE is the "Java Runtime Environment." It only includes the parts of the system that are need to run Java programs. It does not have a compiler. You'll also see the "J2EE SDK." This is the "Enterprise Edition," which includes additional packages that are not needed on a personal computer. *Don't forget to read and follow the installation instructions.*

This textbook is based on Java Version 1.3. If you already have version 1.3, you don't need to download version 1.4 just to use this book.

The Sun site does not have a Java Software Development Kit for Macintosh. However, the Macintosh

OS X operating system already includes Java (Version 1.3 as of July 2002). A Java programming environment is available on the Development CD that comes with OS X. Unfortunately, Java 1.3 is not and will never be available for Macintosh OS 9 and earlier. Java 1.1 can be used on older Macintosh systems, and if you are working on one of those, you might want to use the [previous edition](#) of this book. Information about Java on Macintosh can be found at <http://www.apple.com/java>. For Java programming, see <http://developer.apple.com/java>.

Integrated Development Environments

It is really quite remarkable that there are sophisticated IDEs for Java programming that are available for free. Here are the ones that I know about.

- **Borland JBuilder Personal Edition**, for Linux, Solaris, MacOS X, Windows 2000, Windows XP, and Windows NT. Requires a lot of disk space and memory (256 MB memory recommended). Company Web page at <http://www.borland.com>. JBuilder site at <http://www.borland.com/jbuilder/index.html>. The Personal Edition, which is free, has more than enough features for most programmers.
- **Sun ONE Studio 4 for Java, Community Edition**, for Linux, Solaris, Windows 2000, Windows NT, and Windows 98SE. This was formerly known as "Forte for Java", and you might see it referred under that name. Again, it requires a lot of resources, with a 256 MB memory recommendation. Main site currently at <http://www.sun.com/software/sundev/jde/index.html>. It is available from there and on the J2SE download page, <http://java.sun.com/j2se/1.4/download.html>. The Community Edition is the free version.
- **Mac OS X Project Builder** comes as a standard part of Mac OS X (on the Developer CD). It supports Java as well as some other programming languages.
- **JCreator**, for Windows. I haven't tried it, but it looks like a nice lighter-weight IDE that works on top of Sun's SDK. It was recommended to me by a reader. There is a free version, as well as a shareware version. It is available at <http://www.jcreator.com>.

There are other products similar to JCreator, for Windows and for other operating systems, and you might want to look around if you want some of the convenience of an IDE without all the complexity.

If you want to use any of the sample source code from this book in any of these environments, you will have to figure out how to get the code into the environment. In general, IDEs work with "projects". A project contains the all the source code files needed in the project as well as other information. All this is stored in a project directory. To use a source code file from outside the project, you have to "import" it in some way. Usually, you have to copy the file into the project directory or into a source code directory inside the project directory. In addition to this, you have to use an "Add File" command in the IDE to tell it that the file is part of the project. Details vary from one IDE to another. If all else fails, try using a "New File" command to create an empty window in the IDE, and then copy-and-paste the source code from a web browser window into the IDE's window.

Text Editors

If you decide to use a command-line environment for programming, make sure that you have a good text editor. A programmer's text editor is a very different thing from a word processor. Most important, it saves your work in plain text files and it doesn't insert extra carriage returns beyond the ones you actually type. A good programmer's text editor will do a lot more than this. Here are some features to look for:

- Syntax coloring. Shows comments, strings, keywords, etc., in different colors to make the program easier to read and to help you find certain kinds of errors.
- Function menu. A pop-up menu that lists the functions in your source code. Selecting a function

from this will take you directly to that function in the code.

- **Auto-indentation.** When you indent one line, the editor will indent following lines to match, since that's what you want more often than not when you are typing a program.
- **Parenthesis matching.** When you type a closing parenthesis the cursor jumps back to the matching parenthesis momentarily so you can see where it is. Alternatively, there might be a command that will hilite all the text between matching parentheses. The same thing works for brackets and braces.
- **Indent Block and Unindent Block commands.** These commands apply to a hilited block of text. They will insert or remove spaces at the beginning of each line to increase or decrease the indentation level of that block of text. When you make changes in your program, these commands can help you keep the indentation in line with the structure of the program.
- **Control of tabs.** My advice is, don't use tab characters for indentation. A good editor can be configured to insert multiple space characters when you press the tab key.

There are many free text editors that have some or all of these features. Since you are using Java, you should certainly consider jedit, a programmer's text editor written entirely in Java. It requires Java 1.3 or better. It has many features listed above, and there are plug-ins available to add additional features. Since it is written in pure Java, it can be used on any operating system that supports Java 1.3. In addition to being a nice text editor, it shows what can be done with the Swing GUI. Jedit is free and can be downloaded from <http://www.jedit.org>.

In my own work on Macintosh, I have used BBEdit for Macintosh from Bare Bones Software (<http://www.barebones.com/>). BBEdit is not free, but there is a free version called BBEdit Lite.

On Linux, I generally use nedit. It has all the above features, except a function menu. If you are using Linux, it is likely that *nedit* is included in your distribution, although it may not have been installed by default. It can be downloaded from <http://www.nedit.org/> and is available for many UNIX platforms in addition to Linux. Features such as syntax coloring and auto-indentation are not turned on by default. You can configure them in the Options menu. Use the "Save Options" command to make the configuration permanent. Of course, as alternatives to nedit, the Gnome and KDE desktops for Linux have their own text editors.

Since I have very little experience with Windows, I don't have a recommendation for a programmer's editor for Windows, other than jedit.

Using the Java SDK

If you have installed Sun's Software Development Kit for Java, you can use the commands "javac", "java", and "appletviewer" for compiling and running Java programs and applets. These commands must be on the "path" where the operating system searches for commands. (See the installation instructions on Sun's Java web site.) The rest of this appendix contains some basic instructions for using these commands with this textbook.

I suggest that you make a directory to hold your Java programs. (You might want to have a different subdirectory for each program that you write.) Create your program with a text editor, or copy the program you want to compile into your program directory. If the program needs any extra files, don't forget to get them as well. For example, most of the programs in the early chapters of this textbook require the file [TextIO.java](#). You should copy this file into the same directory with the main program file that uses it. (Actually, you only need the compiled file, `TextIO.class`, to be in the same directory as your program. So, once you have compiled `TextIO.java`, you can just copy the class file to any directories where you need it.)

If you have downloaded a copy of this textbook, you can simply copy the files you need from the [source](#) directory that is part of the download. If you haven't downloaded the textbook, you can open the source file

in a Web browser and then use the Web browser's "Save" command to save a copy of the file. Another way to get Java source code off a Web browser page is to highlight the code on the page, use the browser's "Copy" command to place the code on the Clipboard, and then "Paste" the code into your text editor. You can use this last method when you want to get a segment of code out of the middle of a Web page.

To use the SDK, you will have to work in a command window, using a command-line interface. In Windows, this means a DOS window. In Linux/UNIX, it means an "xterm" or "console" or "terminal" window. Open a command Window and change to the directory that contains your Java source code files. Use the "javac" command for compiling Java source code files. For example, to compile `SourceFile.java`, use the command

```
javac SourceFile.java
```

You must be working in the directory that contains the file. If the source code file does not contain any syntax errors, this command will produce one or more compiled class files. If the compiler finds any syntax errors, it will list them. Note that not every message from the javac compiler is an error. In some cases, it generates "warnings" that will not stop it from compiling the program. If the compiler finds errors in the program, you can edit the source code file and try to compile it again. Note that you can keep the source code file open in a text editor in one window while you compile the program in the command window. Then, it's easy to go back to the editor to edit the file. However, when you do this, don't forget to save the modifications that you make to the file before you try to compile it again! (Some text editors can be configured to issue the compiler command for you, so you don't even have to leave the text editor to run the compiler.)

If your program contains more than a few errors, most of them will scroll out of the window before you see them. In Linux and UNIX, a command window usually has a scroll bar that you can use to review the errors. In Windows 2000/NT/XP (but not Windows 95/98), you can save the errors in a file which you can view later in a text editor. The command in Windows is

```
javac SourceFile.java >& errors.txt
```

The ">& errors.txt" redirects the output from the compiler to the file, instead of to the DOS window. For Windows 95/98 I've written a little Java program that will let you do much the same thing. See the source code for that program, [cef.java](#), for instructions.

It is possible to compile all the Java files in a directory at one time. Use the command "javac *.java".

(By the way, all these compilation commands only work if the classes you are compiling are in the "default package". This means that they will work for any example from this textbook. But if you start defining classes in other packages, the source files must be in subdirectories with corresponding names. For example, if a class is in the package named `utilities.drawing` then the source code file should be in a directory named `drawing`, which is in a directory named `utilities`, which is in the top-level program directory. You should work in the top-level directory and compile the source code file with a command such as `javac utilities/drawing/sourcefile.java` on Linux/UNIX or `javac utilities\drawing\sourcefile.java` on Windows. If you don't do it like this, the compiler might not be able to find other classes that your class uses.)

Once you have your compiled class files, you are ready to run your application or applet. If you are running a stand-alone application -- one that has a `main()` routine -- you can use the "java" command from the SDK to run the application. If the class file that contains the `main()` routine is named `Main.class`, then you can run the program with the command:

```
java Main
```

Note that this command uses the name of the class, "Main", not the full name of the class file, "Main.class". This command assumes that the file "Main.class" is in the current directory, and that any other class files used by the main program are also in that directory. You do not need the Java source code files to run the

program, only the compiled class files. (Again, all this assumes that the classes you are working with are in the "default package". Classes in other packages should be in subdirectories.)

If your program is an applet, then you need an HTML file to run it. See [Section 6.2](#) for information about how to write an HTML file that includes an applet. As an example, the following code could be used in an HTML file to run the applet "MyApplet.class":

```
<applet code="MyApplet.class" width=300 height=200>
</applet>
```

The "appletviewer" command from the SDK can then be used to view the applet. If the file name is `test.html`, use the command

```
appletviewer test.html
```

This will only show the applet. It will ignore any text or images in the HTML file. In fact, all you really need in the HTML file is a single applet tag, like the example shown above. The applet will be run in a resizable window, but you should remember that many of the applet examples in this textbook assume that the applet will not be resized. Note also that your applet can use standard output, `System.out`, to write messages to the command window. This can be useful for debugging your applet.

You can use the appletviewer command on any file, or even on a web page address. It will find all the applet tags in the file, and will open a window for each applet. If you are using a Web browser that does not support Java 2, you could use appletviewer to see the applets in this book. For example, to see the applets in Section 6.1, use the command

```
appletviewer http://math.hws.edu/javanotes/c6/s1.html
```

to view the applets directly off the web. Or, if you have downloaded the textbook, you can change to the directory `c6` and use the command `appletviewer s1.html` to see the applets.

Of course, it's also possible to view your own applets in a Web browser. Just open the html file that contains the applet tag for your applet. One problem with this is that if you make changes to the applet, you might have to actually quit the browser and restart it in order to get the changes to take effect. The browser's Reload command might not cause the modified applet to be loaded.

[[Main Index](#)]

Introduction to Programming Using Java, Fourth Edition

Source Code

THIS PAGE CONTAINS LINKS to the source code for examples appearing in the free, on-line textbook [Introduction to Programming Using Java](http://math.hws.edu/javanotes/), Version 4.0, which is available at <http://math.hws.edu/javanotes/>. You should be able to compile these files and use them. Note however that some of these examples depend on other classes, such as `TextIO.class` and `MosaicFrame.class`, that are not built into Java. These are classes that I have written. Links to the source code are provided below. To use examples that depend on my classes, you will need to compile the source code for the required classes and place the compiled classes in the same directory with the main class file. If you are using an integrated development environment such as CodeWarrior or JBuilder, you can simply add any required source code files to your project. See [Appendix 2](#) for more information on Java programming environments and how to use them to compile and run these examples.

Most of the solutions to end-of-chapter exercises are **not listed on this page**. Each end-of-chapter exercise has its own Web page, which discusses its solution. The source code of a sample solution of each exercise is given in full on the solution page for that exercise. If you want to compile the solution, you should be able to cut-and-paste the solution out of a Web browser window and into a text editing program. (You can't cut-and-paste from the HTML source of the solution page, since it contains extra HTML markup commands that the Java compiler won't understand.)

Note that many of these examples require Java version 1.3 or later. Some of them were written for older versions, but will still work with current versions. When you compile some of these older programs with current versions of Java, you might get warnings about "deprecated" methods. These warnings are not errors. When a method is deprecated, it means that it should not be used in new code, but it has not yet been removed from the language. It is possible that deprecated methods might be removed from the language at some future time, but for now you just get a warning about using them.

Part 1: Text-oriented Examples

Many of the sample programs in the text are based on console-style input/output, where the computer and the user type lines of text back and forth to each other. Some of these programs use the standard output object, `System.out`, for output. Most of them use my non-standard class, `TextIO` for both input and output. The programs are stand-alone applications, not applets, but I have written applets that simulate many of the programs. These "console applets" appear on the Web pages that make up the text. The following list includes links to the source code for each applet, as well as links to the source code of the programs that the applets simulate. All of the console applets depend on classes defined in the files [ConsoleApplet.java](#), [ConsolePanel.java](#), and [ConsoleCanvas.java](#). These three files, or the class files compiled from them, must be available when you compile any console applet. The class files must be available when the applet is used. (Being "available" means being in the same directory where you are compiling the program, or being in the same directory as the HTML file that uses the applet.) Most of the standalone programs depend on the `TextIO` class, which is defined in [TextIO.java](#). Either `TextIO.java` or `TextIO.class` must be available when you compile the program, and `TextIO.class` must be available when you run the program. These programs and applets will work with Java 1.1, as well as with later versions.

- [ConsoleApplet.java](#), a basic class that does the HelloWorld program in [Section 2.1](#). (The other console applets, below, are defined as subclasses of `ConsoleApplet`.)
- [Interest1Console.java](#), the first investment example, from [Section 2.2](#). Simulates [Interest.java](#).

- [TimedComputationConsole.java](#), which does some simple computations and reports how long they take, from [Section 2.3](#). Simulates [TimedComputation.java](#).
- [PrintSquareConsole.java](#), the first example that does user input, from [Section 2.4](#). Simulates [PrintSquare.java](#).
- [Interest2Console.java](#), the second investment example, with user input, from [Section 2.4](#). Simulates [Interest2.java](#).
- [Interest3Console.java](#), the third investment example, from [Section 3.1](#). Simulates [Interest3.java](#).
- [ThreeN1Console.java](#), the "3N+1" program from [Section 3.2](#). Simulates [ThreeN1.java](#)
- [ComputeAverageConsole.java](#), which finds the average of numbers entered by the user, from [Section 3.3](#). Simulates [ComputeAverage.java](#)
- [CountDivisorsConsole.java](#), which finds the number of divisors of an integer, from [Section 3.4](#). Simulates [CountDivisors.java](#)
- [ListLettersConsole.java](#), which lists all the letters that occur in a line of text, from [Section 3.4](#). Simulates [ListLetters.java](#)
- [LengthConverterConsole.java](#), which converts length measurements between various units of measure, from [Section 3.5](#). Simulates [LengthConverter.java](#)
- [PrintProduct.java](#), which prints the product of two numbers from [Section 3.7](#). (This was given as an example of writing console applets, and it does not simulate any stand-alone program example.)
- [GuessingGameConsole.java](#), the guessing game from [Section 4.2](#). Simulates [GuesingGame.java](#). A slight variation of this program, which reports the number of games won by the user, is [GuesingGame2.java](#).
- [RowsOfCharsConsole.java](#), a useless program that illustrates subroutines from [Section 4.3](#). Simulates [RowsOfChars.java](#).
- [TheeN2Console.java](#), an improved 3N+1 program from [Section 4.4](#). Simulates [ThreeN2.java](#)
- [RollTwoPairsConsole.java](#) rolls two pairs of dice until the totals come up the same, from [Section 5.2](#). Simulates [RollTwoPairs.java](#). The applet and program use the class [PairOfDice.java](#).
- [HighLowConsole.java](#) plays a simple card game, from [Section 5.3](#). Simulates [HighLow.java](#). The applet and program use the classes [Card.java](#) and [Deck.java](#). (The `Deck` class uses arrays, which are not covered until [Chapter 8](#).)
- [BlackjackConsole.java](#) lets the user play a game of Blackjack, [from the exercises for Chapter 5](#). Uses the classes [Card.java](#), [Hand.java](#), [BlackjackHand.java](#) and [Deck.java](#).
- [BirthdayProblemConsole.java](#) is a small program that uses arrays, from [Section 8.2](#). Simulates [BirthdayProblemDemo.java](#).
- [ReverseIntsConsole.java](#) demonstrates a dynamic array of ints by printing a list of input numbers in reverse order, from [Section 8.3](#). Simulates [ReverseWithDynamicArray.java](#), which uses the dynamic array class defined in [DynamicArrayOfInt.java](#). A version of the program that uses an ordinary array of ints is [ReverseInputNumbers.java](#).
- [LengthConverter2Console.java](#), an improved version of [LengthConverterConsole.java](#). It converts length measurements between various units of measure. From [Section 9.2](#). Simulates [LengthConverter2.java](#)
- [LengthConverter3.java](#) is a version of the previous program, [LengthConverter2.java](#), which uses exceptions to handle errors in the user's input. From the user's point of view, the behavior of `LengthConverter3` is identical to that of `LengthConverter2`, so I didn't include an applet version in

the text. From [Section 9.4](#).

- [ReverseFile.java](#), a program that reads a file of numbers and writes another file containing the same numbers in reverse order. From [Section 10.2](#). This file depends on [TextReader.java](#). Since applets cannot manipulate files, there is no applet version of this program.
- [WordList.java](#), a program that makes a list of the words in a file and outputs the words to another file. From [Section 10.3](#). Depends on [TextReader.java](#). There is no applet version of this program.
- [CopyFile.java](#), a program that copies a file. The input and output files are specified as command line arguments. From [Section 10.3](#). There is no applet version of this program.
- Two pairs of command-line client/server network applications from [Section 10.5](#): [DateServe.java](#) and [DateClient.java](#); [CLChatServer.java](#) and [CLChatClient.java](#). There are no corresponding applets.
- [TowersOfHanoiConsole.java](#), a console applet that gives a very simple demonstration of recursion, from [Section 11.1](#).
- [ListDemoConsole.java](#) demonstrates the list class that is defined in [StringList.java](#), from [Section 11.2](#). Simulates [ListDemo.java](#).
- [PostfixEvalConsole.java](#) uses a stack to evaluate postfix expressions, from [Section 11.3](#). The stack class is defined in [NumberStack.java](#). Simulates [PostfixEval.java](#).
- [SortTreeConsole.java](#) demonstrates some subroutines that process binary sort trees, from [Section 11.4](#). Simulates [SortTreeDemo.java](#).
- [SimpleParser3Console.java](#) reads expressions entered by the user and builds expression trees to represent them. From [Section 11.5](#). Simulates [SimpleParser3.java](#). Related programs, which evaluate expressions without building expression trees, are [SimpleParser1.java](#) and [SimpleParser2.java](#).
- [ListInsert.java](#), a very short program that demonstrates a subroutine for inserting a new item into a sorted generic List, from [Section 12.2](#). There is no corresponding Console applet.
- [WordListWithTreeSet.java](#), another demonstration program from [Section 12.2](#). It makes a list of distinct words from a file. This is a version of [WordList.java](#) that uses a `TreeSet` to store the words. There is no corresponding Console applet.
- [SimpleParser5Console.java](#) uses a `HashMap` as a symbol table in a program that can evaluate expressions that contain variables, from [Section 12.4](#). This applet simulates the program [SimpleParser5.java](#).
- [WordCount.java](#) uses Maps, Sets, and Lists to make a list of all the words in a file along with the number of times that each word occurs in the file, from [Section 12.4](#). This program requires [TextReader.java](#). There is no applet version.

Part 2: Graphical Examples from the Text

- [GUIDemo.java](#) and [GUIDemo2.java](#), simple GUI demo applets from [Section 1.6](#). These applets demonstrate AWT and Swing components, respectively. (You won't be able to understand the source code until you read Chapters 6 and 7.)
- [StaticRects.java](#), a rather useless applet from [Section 3.7](#) that just draws a set of nested rectangles.
- [MovingRects.java](#), the sample animation applet from [Section 3.7](#). (This depends on [SimpleAnimationApplet2.java](#).)

- [RandomMosaicWalk.java](#), a standalone program that displays a window full of colored squares with a moving disturbance, from [Section 4.6](#). (This depends on [MosaicCanvas.java](#) and [Mosaic.java](#).) The applet version of the random walk, which is shown on the web page, is [RandomMosaicWalkApplet.java](#). The source code for the applet uses some advanced techniques.
- [RandomMosaicWalk2.java](#) is a version of the previous program, [RandomMosaicWalk.java](#), modified to use a few named constants. From [Section 4.7](#).
- [ShapeDraw.java](#), the applet with draggable shapes, from [Section 5.4](#). This file produces six class files when it is compiled. You won't be able to understand everything in this file until you've read Chapters 6 and 7.
- [HelloWorldApplet.java](#) and [HelloWorldApplet2.java](#), the utterly basic first sample applet, from [Section 6.1](#). The second version has an `init()` method to set its foreground and background colors.
- [HelloSwing.java](#) and [HelloSwing2.java](#), a very basic sample applet using Swing, events, and a dialog box, from [Section 6.1](#). The second version uses an anonymous nested class to respond to clicks on a button.
- [ColorChooserApplet.java](#), an applet for investigating RGB and HSB colors. This is a Java 1.1 applet which uses the AWT rather than Swing. From [Section 6.3](#).
- [RandomStrings.java](#), which draws randomly colored and positioned strings, from [Section 6.3](#).
- [ClickableRandomStrings.java](#), an extension of the previous applet in which the applet is redrawn when the user clicks it with the mouse, from [Section 6.4](#). ([ClickableRandomStrings2.java](#) is an equivalent class that uses an anonymous subclass of `MouseAdapter` to do the event handling.)
- [SimpleStamper.java](#), a basic demo of `MouseEvent`s, from [Section 6.4](#).
- [SimpleTrackMouse.java](#), which displays information about mouse events, from [Section 6.4](#).
- [SimplePaint.java](#), a first attempt at a paint program in which the user can select colors and draw curves, from [Section 6.4](#).
- [KeyboardAndFocusDemo.java](#), which demos keyboard events, from [Section 6.5](#).
- [SubKillerGame.java](#), a simple arcade-style game, from [Section 6.5](#). This applet is based on [KeyboardAnimationApplet2.java](#).
- [HelloWorldJApplet.java](#), a fairly simple example of using layouts and multiple buttons, from [Section 6.6](#).
- [HighLowGUI.java](#), a simple card game, from [Section 6.5](#). This file defines two classes used by the applet. The program also depends on [Card.java](#), [Hand.java](#), and [Deck.java](#).
- [SimplePaint2.java](#), a second attempt at a paint program in which the user can select colors and draw curves, from [Section 6.5](#). This file defines two classes that are used by the applet.
- [HighLowGUI2.java](#), a version of the simple card game, [HighLowGUI.java](#). This version gets pictures of cards from an image file. From [Section 7.1](#).
- [DoubleBufferedDrag.java](#) and [NonDoubleBufferedDrag.java](#), two little applets that demonstrate double buffering. In the second, double buffering is turned off. From [Section 7.1](#).
- [RubberBand.java](#), a little applet that illustrates using an off-screen image and rubber band cursor, from [Section 7.1](#).
- [SimplePaint3.java](#), an improved paint program that uses an off-screen canvas to back up the on-screen image, from [Section 7.1](#).
- [LayoutDemo.java](#), which demos a variety of layout managers, from [Section 7.2](#).

- [BorderDemo.java](#), which shows six different type of Borders, from [Section 7.2](#).
- [RGBColorChooser.java](#), a simplified version of [ColorChooserApplet.java](#) that lets the user select a color with three sliders that control the RGB components, from [Section 7.4](#).
- [SimpleCalculator.java](#), which lets the user do arithmetic operations using JTextFields and JButtons, from [Section 7.4](#).
- [StopWatch.java](#) and [MirrorLabel.java](#), two small custom component classes, and [ComponentTest.java](#), an applet that tests them. From [Section 7.4](#).
- [NullLayoutDemo.java](#), which demonstrates how to do your own component layout instead of using a layout manager, from [Section 7.4](#).
- [ShapeDrawWithMenus.java](#), an improved version of [ShapeDraw.java](#) that uses a menu bar, menus, and a pop-up menu, from [Section 7.5](#).
- [HelloWorldSpectrum.java](#), an applet that displays the message "HelloWorld" in animated color, from [Section 7.6](#). A first example of using a `Timer` directly to animate an applet.
- [ScrollingHelloWorld.java](#), an applet that scrolls a message, from [Section 7.6](#). Shows how to animate an applet with a `Timer` created in the applet's `start()` method.
- [Mandelbrot.java](#), an applet that draws a representation of the famous Mandelbrot set, from [Section 7.6](#). This applet creates a separate thread to do the long computation of the colors for the set.
- [ShapeDrawFrame.java](#), another version of [ShapeDraw](#) as a `JFrame` instead of an `JApplet`. From [Section 7.7](#). The `ShapeDrawFrame` class contains a `main()` routine and can be run as an application. The applet [ShapeDrawLauncher.java](#), merely displays a button. When you click on the button, a `ShapeDrawFrame` window is opened.
- [HighLowFrame.java](#), another version of [HighLowGUI2](#) as a `JFrame` instead of an `JApplet`, and with a main program to run it as an application. The applet [HighLowLauncher.java](#) is a button that can be used to open a `HighLowFrame` window.
- [SimpleDialogDemo.java](#), a little applet that just demonstrates four of Swing's standard dialog boxes. From [Section 7.7](#).
- [RandomStringsFromArray.java](#), which draws randomly colored and positioned strings and uses an array to remember what it has drawn, from [Section 8.2](#).
- [SimpleDrawRects.java](#), in which the user can place colored rectangles on a canvas and drag them around, from [Section 8.3](#). This simplified shape-drawing program is meant to illustrate the use of vectors. The file also defines a reusable custom component, `RainbowPalette`.
- [Checkers.java](#), which lets two people play checkers against each other, from [Section 8.5](#). At 702 lines, this is a relatively large program.
- [TrivialEdit.java](#), a standalone application which lets the user edit short text files, from [Section 10.3](#). This program depends on [TextReader.java](#).
- [ShapeDrawWithFiles.java](#), a final version of [ShapeDraw.java](#) that uses files to save and reload the designs created with the program. This version is an independent program, not as an applet. It is described at the end of [Section 10.3](#).
- [ReadURLApplet.java](#), an applet that reads data from a URL, from [Section 10.4](#). This is similar to the command-line program [ReadURL.java](#), from the same section.
- [ChatSimulation.java](#), an that simulates a two-way network connection, using a thread, from [Section 10.5](#).
- [ChatWindow.java](#), a `JFrame` that supports chatting between two users over the network, from

[Section 10.5](#). This class depends on [TextReader.java](#). This class can be run as a standalone application, as either a server or a client.

- [BrokeredChat.java](#), an applet that sets up chat connections that use the previous example, [ChatWindow.java](#). There is a server program, [ConnectionBroker.java](#), which must be running on the computer from which the Web page containing the applet was downloaded. (The server keeps a list of available "chatters" for the applet.) From [Section 10.5](#).
 - [Blobs.java](#), an applet that demonstrates recursion, from [Section 11.1](#).
 - [DepthBreadth.java](#), an applet that uses stacks and queues, from [Section 11.3](#).
-

Part 3: End-of-Chapter Applets

This section contains the source code for the applets that are used as decorations at the end of each chapter. In general, you should not expect to be able to understand these applets at the time they occur in the text. Most of these are older applets will work with Java 1.1 or even Java 1.0.

1. [Moire.java](#), an animated design, shown at the end of [Section 1.7](#). (You can use applet parameters to control various aspects of this applet's behavior. Also note that you can click on the applet and drag the pattern around by hand. See the source code for details.)
2. [JavaPops.java](#), and applet that shows multi-colored "Java!"s, from the end of [Section 2.5](#). (This depends on [SimpleAnimationApplet.java](#).)
3. [MovingRects.java](#), the sample animation applet from [Section 3.7](#). (This depends on [SimpleAnimationApplet2.java](#).) This is also listed above, as one of the graphical examples from the text.
4. [RandomBrighten.java](#), showing a grid of colored squares that get more and more red as a wandering disturbance visits them, from the end of [Section 4.7](#). (Depends on [MosaicCanvas.java](#).) (Another applet that shows an animation based on [MosaicCanvas.java](#) is [MosaicStrobeApplet.java](#), the applet version of the solution to one of the [exercises for Chapter 4](#).)
5. [SymmetricBrighten.java](#), a subclass of the previous example that makes a symmetric pattern, from the end of [Section 5.6](#). Depends on [MosaicCanvas.java](#) and [RandomBrighten.java](#).
6. [TrackLines.java](#), an applet with lines that track the mouse, from [Section 6.6](#).
7. [KaleidaAnimate.java](#), an applet that shows symmetric, kaleidoscope-like animations, from [Section 7.7](#). Depends on [SimpleAnimationApplet.java](#).
8. [Maze.java](#), an applet that creates a random maze and solves it, from [Section 8.5](#).
9. [SimpleCA.java](#), a Cellular Automaton applet, from the end of [Section 9.4](#). This applet depends on the file [CACanvas.java](#). For more information on cellular automata see <http://math.hws.edu/xJava/CA/>.
10. [TowersOfHanoi.java](#), an animation of the solution to the Towers of Hanoi problem for a tower of ten disks, from the end of [Section 10.5](#).
11. [LittlePentominosApplet.java](#), the pentominos applet from the end of [Section 11.5](#). This file defines two classes, [LittlePentominosApplet](#) and [PentominosBoardCanvas](#). A pentomino is made up of five connected squares. This applet solves puzzles that involve filling a board with pentominos. If you click on the applet it will start a new puzzle. For more information see <http://math.hws.edu/eck/xJava/PentominosSolver/> where you'll also find the big brother of this little applet.
12. The applet at the end of [Section 12.4](#) is the same [Moire.java](#) that was used at the end of Chapter 1.

Part 4: Required Auxiliary Files

This section lists many of the extra source files that are required by various examples in the previous sections, along with a description of each file. The files listed here are those which are general enough to be useful in other programming projects.

- [TextIO.java](#) which defines a class containing some static methods for doing input/output. These methods make it easier to use the standard input and output streams, `System.in` and `System.out`. The `TextIO` class defined in this file will be useless on a system that does not implement standard input. In that case, try using the following file instead.
- [TextIO.java for GUI](#) defines an alternative version of the `TextIO` class. It defines the same set of input and output routines as the original version of `TextIO`. But instead of using standard I/O, it opens its own window, and all the input/output is done in that window. Please read the comments at the beginning of the file. (For people who have downloaded this book: The file is located in a directory named `TextIO-GUI` inside the source directory.)
- [ConsoleApplet.java](#), a class that can be used as a framework for writing applets that do console-style input/output. To write such an applet, you have to define a subclass of `ConsoleApplet`. See the source code for details. Many examples of applets created using `ConsoleApplet` are available above. Any project that uses this class also requires [ConsolePanel.java](#) and [ConsoleCanvas.java](#).
- [ConsolePanel.java](#), a support class that is required by any project that uses `ConsoleApplet`.
- [ConsoleCanvas.java](#), a support class that is required by any project that uses `ConsoleApplet`.
- [SimpleAnimationApplet2.java](#), a class that can be used as a framework for writing animated applets. To use the framework, you have to define a subclass of `SimpleAnimationApplet`. This class uses Swing and requires Java 1.3 or higher. [Section 3.7](#) has an example.
- [SimpleAnimationApplet.java](#), a class that can be used as a framework for writing animated applets. This class has the same functionality as the previous class, but it is a Java 1.0 applet and so can be used even with very old versions of Java. This file is used as the basis for some of my end-of-chapter applets.
- [Mosaic.java](#) which let's you write programs that work with a window full of rows and columns of colored rectangles. `Mosaic.java` depends on [MosaicCanvas.java](#). There is an example in [Section 4.6](#).
- [MosaicCanvas.java](#), a subclass of the built-in `Canvas` class that implements a grid of colored rectangles.
- [KeyboardAnimationApplet2.java](#), a class that can be used as a framework for writing animated applets, which the user can interact with by using the keyboard. This framework can be used for simple arcade-style games, such as the SubKiller game in [Section 6.5](#). To use the framework, you have to define a subclass of `KeyboardAnimationApplet2`. This requires Java 1.2 or higher.
- [KeyboardAnimationApplet.java](#), an older version of the previous class that has essentially the same functionality but that works with Java 1.1. (This version is not used in this textbook.)
- [Expr.java](#), a class for working with mathematical expressions that can include the variable `x` and mathematical functions such as `sin` and `sqrt`. This class was used in [Exercise 9.4](#).
- [TextReader.java](#), a class that can be used to read data from text files and other input streams. From [Section 10.1](#).

David Eck (eck@hws.edu), July 2002

GNU Free Documentation License
Version 1.1, March 2000

Copyright (C) 2000 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed,

as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated

as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified

Version as stated in the previous sentence.

- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by

adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except

as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (c)  YEAR  YOUR NAME.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.1
or any later version published by the Free Software Foundation;
with the Invariant Sections being LIST THEIR TITLES, with the
Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.
A copy of the license is included in the section entitled "GNU
Free Documentation License".
```

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.