

Introduction to Programming Using Java Version 3.1, February 2001

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 the Java applets that are embedded throughout the text. In most places where an applet should appear, you will see a message such as "Sorry, but your Web browser does not support Java." Also not included are Java source code examples from Appendix 3 of the text and solutions to the quizzes and programming exercises. The real version of the textbook is on-line, to be read with a Web browser. Version 3.1 contains only minor corrections from Version 3.0, which was released in May 2000.

**Permission is hereby granted to duplicate, modify,
and distribute all or part of the following material,
under the terms of the [Open Publication License](#).**

Introduction to Programming Using Java

Version 3.1, February 2001

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

WELCOME TO *Introduction to Programming Using Java*, an on-line textbook on introductory programming, which uses Java as the language of instruction. This text has more than enough material for a one-semester course, and it also suitable for individuals who want to learn programming on their own. This is the third edition of the text. (Version 3.1 is a minor upgrade of Version 3.0, which was released in May, 2000. Version 3.1 incorporates a few changes and corrections, and it is released under the [Open Publication License](#).) The third edition covers more material and has more examples than the second edition. It also adds end-of-chapter quizzes and solved programming exercises. Previous editions have been used in a course, *Computer Science 124: Introductory Programming*, at [Hobart and William Smith Colleges](#). (The title of the previous editions included a reference to this course.) This textbook covers Java 1.1. Most of the applets that are contained in the text require Java 1.1 or higher.

Links for downloading copies of this text can be found at the bottom of this page. To learn more about this on-line text, please read its [preface](#).

Search this Text:

Although this book does not have a conventional index, you can search it for terms that interest you. Note that this searches the version of the text book at its main site, at math.hws.edu.

Search Introduction to Programming Using Java for pages...

Short Table of Contents:

- [Full Table of Contents](#)
- [Preface](#)
- [Preface to the Second Edition](#)
- 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: [Components and Events](#)
 - Chapter 8: [Arrays](#)
 - Chapter 9: [Correctness and Robustness](#)
 - Chapter 10: [Advanced Input/Output](#)
 - Chapter 11: [Linked Data Structures and Recursion](#)
 - Appendix 1: [From Java to C++](#)
 - Appendix 2: [Some Notes on Java Programming Environments](#)
 - Appendix 3: [Source code for all examples in the text](#)
 - [News and Errata](#)
-

This is a free textbook. As of Version 3.1, it is published under the terms of the [Open Publication License](#), Version 1.0. The latest edition is always available, at no charge, for downloading and for on-line use at the Web address <http://math.hws.edu/javanotes/>. This edition, the third, is also permanently archived at the address <http://math.hws.edu/eck/cs124/javanotes3/>. A copy can also be found at Andamooka.org, where readers can post annotations and read other user's annotations.

Downloading Links

Use one of the following direct links to download a compressed archive of this entire textbook. You can use this material on your own computer. You can also re-post it on any Web server. See the [preface](#) for more information on usage restrictions and for full downloading instructions. A PDF file that can be used for printing the textbook is also available there.

- <http://math.hws.edu/eck/cs124/downloads/javanotes3.zip> (1.6 MB), for Windows.
- <http://math.hws.edu/eck/cs124/downloads/javanotes3.sit.hqx> (2.1 MB), for Macintosh.
- <http://math.hws.edu/eck/cs124/downloads/javanotes3.tar.Z> (1.8 MB), for Linux/UNIX.

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

Version 3.0, May 2000

Version 3.1, with minor changes, February 2001

Introduction to Programming Using Java, Third Edition

Table of Contents

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

[Preface](#)

[Preface to the Second Edition](#)

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: [More Details of Classes](#)
- [Programming Exercises](#)
- [Quiz on this Chapter](#)

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

- Section 1: [The Basic Java Applet](#)
- Section 2: [HTML Basics and the Web](#)
- Section 3: [Graphics and the Paint Method](#)
- Section 4: [Mouse Events](#)
- Section 5: [Keyboard Events](#)
- Section 6: [Introduction to Layouts and Components](#)
- Section 7: [Looking Back: The Java 1.0 Event Model](#)
- [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: [Standard Components and Their Events](#)
- Section 4: [Programming with Components](#)
- Section 5: [Threads, Synchronization, and Animation](#)
- Section 6: [Nested Classes and Adapter Classes](#)
- Section 7: [Frames and Dialogs](#)
- Section 8: [Looking Forward: Swing and Java 2.0](#)
- [Programming Exercises](#)
- [Quiz on this Chapter](#)

Chapter 8: [Arrays](#)

- Section 1: [Creating and Using Arrays](#)
- Section 2: [Programming with Arrays](#)
- Section 3: [Vectors and Dynamic Arrays](#)
- 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](#)

Appendix 1: [From Java to C++](#)

- [Section 1: C++ Programming Fundamentals](#)
- [Section 2: Pointers and Arrays in C++](#)
- [Section 3: Classes and Objects in C++](#)

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), May 2000

Introduction to Programming Using Java, Third Edition (Version 3.1)

Preface

"INTRODUCTION TO PROGRAMMING WITH JAVA" is a free, on-line textbook. It is suitable for use in an introductory programming course and for people who are trying to learn programming on their own. There are no prerequisites beyond a general familiarity with the ideas of computers and programs.

This text uses the Java programming language as the language of instruction. It requires Java version 1.1 or higher. In style, this is a textbook rather than a tutorial. That is, it concentrates on explaining concepts rather than giving step-by-step how-to-do-it guides. It is certainly not a Java reference book, and it is not even a comprehensive survey of all the features of Java. It is not a quick introduction to Java for people who already know another programming language. Instead, it is directed mainly towards people who are learning programming for the first time, and it is as much about general programming concepts as it is about Java in particular.

This is the third edition of *Introduction to Programming with Java*. The first two editions have been used by the author and by another professor in the introductory programming class at Hobart and William Smith Colleges (<http://www.hws.edu/>). The new edition is a major upgrade. It is more than twice the size of the second edition. Changes include:

- [Chapter 11](#), on linked data structures and recursion, is completely new. [Chapter 9](#), on correctness and robustness, is new except for the section on the `try...catch` statement.
- A single chapter on "programming in the small" from the previous edition has been expanded to two chapters ([Chapter 2](#) and [Chapter 3](#)) in this edition.
- Every chapter, except the first, now includes a set of programming exercises. A solution is provided for each exercise, along with a discussion of the programming involved.
- There is a sample quiz at the end of each chapter, with answers.
- Many sections from the previous edition have been rewritten, and many new examples have been added. As in the previous editions, the source code for every example is included in [an appendix](#).
- Based on experience with the previous editions, the exposition of some topics has been modified by postponing certain details until later in the text. This is especially true in the two chapters on graphical user interface programming ([Chapter 6](#) and [Chapter 7](#) in this edition). These chapters have been completely reorganized.

With these changes, *Introduction to Programming with Java* is now fully competitive, in the author's opinion, with the conventionally published, printed programming textbooks that are available on the market. (Well, all right, I'll confess that I think it's better.)

This textbook differs from many other Java programming books in that it does not deal primarily with applets. Early chapters concentrate on standalone applications that use text input and output. Applets are introduced briefly in [Section 3.7](#) and covered pretty thoroughly in [Chapter 6](#) and [Chapter 7](#). In the remaining chapters, applets are used in many but not all examples and exercises. "Swing," a new set of interface components introduced in Java 1.2, is just barely mentioned (in [Section 7.8](#)). This approach allows a gentler introduction to fundamental programming concepts, and it postpones the complexities of graphical user interface programming until a time when students are ready to deal with them. The decision to do things this way also reflects the fact that applets are only one aspect of Java, and probably not the most important.

I do not plan any further major upgrades to this textbook, but I will probably release new versions in the future with minor revisions and corrections. The current edition of *Introduction to Programming with Java* will always be available at the following Web address:

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

Version 3.1 (February 2001) is a minor upgrade to Version 3.0 (May 2000). It incorporates corrections to a few errors in Version 3.0. (See the [Version 3.0 errata page](#) for a list.) The major change is that with Version 3.1, modification and republication is now covered by the terms of the [Open Publication License](#).

The first, second, and third editions are permanently archived at the following addresses:

First edition: <http://math.hws.edu/eck/cs124/javanotes1/>

Second edition: <http://math.hws.edu/eck/cs124/javanotes2/>

Third edition: <http://math.hws.edu/eck/cs124/javanotes3/>

Downloading the Text

The complete *Introduction to Programming with Java* is available for download as a compressed archive for the Windows, Macintosh, or Linux/Unix platforms. (Text files have slightly different formats on the three platforms. The text files in each archive are in the appropriate format for the platform. For many purposes, though, the difference is unimportant. For example, Web browsers will accept files in any of the formats.) The uncompressed archives contain 580 files and directories and take up over four megabytes of space. You should be able to download an archive by clicking on one of the following links. If you have problems with the downloading, please let me know!

- <http://math.hws.edu/eck/cs124/downloads/javanotes3.zip> (1.6 MB), for Windows.
- <http://math.hws.edu/eck/cs124/downloads/javanotes3.sit.hqx> (2.1 MB), for Macintosh.
- <http://math.hws.edu/eck/cs124/downloads/javanotes3.tar.Z> (1.8 MB), for Linux/UNIX.

An archive must be uncompressed to be useful. To do this, you will need appropriate software (which might already be on your computer). For Windows, you can use WinZip, available from www.winzip.com. WinZip is shareware, but you can use it for a 30 day trial without charge. Alternatively, you might want to get the free program, Aladdin Expander for Windows from www.aladdinsys.com, which can also be used to uncompress the Windows archive. For Macintosh, you need Stuffit Expander for Macintosh, which is already included with most Web browsers. In fact, your Web browser might uncompress the archive automatically when you download it. If you don't have it, Stuffit Expander can be downloaded from www.aladdinsys.com. The software for Linux/UNIX should already be included on your system. To decode the archive javanotes3.tar.Z, use the command "uncompress javanotes3.tar.Z" followed by the command "tar xf javanotes3.tar".

I recommend reading *Introduction to Programming with Java* with a Web browser, so that you can see and use the applets that occur throughout the text. However, I know from experience that a lot of people will want to print all or part of the text. To make this a little easier, I've made a large PDF file that contains the entire textbook, except for the Java source code files from Appendix 3 and the solutions to the quizzes and programming exercises. Of course, the PDF file does not display the applets in the text. Where they should appear, you'll generally see a message such as "Sorry, but your browser does not support Java." A PDF file can be viewed or printed using the free program, Adobe Acrobat Reader. (The file was created using the "Web Capture" feature in Adobe Acrobat Pro 4.0. This is nothing fancy -- just all the Web pages captured in a single file.) The PDF file is available through the following link. It is more than 1.8 megabytes in size, and it contains more than 500 pages of text.

- <http://math.hws.edu/eck/cs124/downloads/javanotes3.pdf> (1.8 MB)

If there is a PDF viewer built into your browser, clicking on the above link will show the file in your Web browser window. In that case, to download the file, try right-clicking or Control-clicking the link. This should bring up a menu that contains a command such as "Save this link". Selecting that command will allow you to download the file to your hard disk.

Usage Restrictions

Introduction to Programming with Java is free, but it is not in the public domain. As of Version 3.1, it is published under the terms of the [Open Publication License](#). (For the purpose of this license, I am both the publisher and author of the work.) This license allows redistribution and modification under certain terms. For example, you can:

- Post an unmodified copy of this textbook on your own Web site.
- Give away or sell printed, unmodified copies of this book, as long as they meet the requirements of the license.
- Post on the web or otherwise distribute modified copies, provided that the modifications are clearly noted in accordance with the license.

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

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

May 23, 2000
Modified February 18, 2001

[[Main Index](#)]

Introduction to Programming Using Java

Preface to the Previous Edition, Fall 1998

"INTRODUCTION TO PROGRAMMING WITH JAVA" is the "second edition" of an on-line introductory programming textbook that uses Java as the language of instruction. This book does not claim to cover the Java language comprehensively, although it does cover enough of it to make it possible to write interesting programs and applets. The main point of the text, however, is to teach the basics of programming -- including object-oriented programming -- with no prerequisites except a general familiarity with the ideas of computers and programs.

This text should be useful to anyone who wants to learn Java, but who is not already an expert in C and C++. Unlike many introductions to Java programming, it does not assume any background in these languages.

Many working applets are included on the Web pages that make up the text, and the full source code for all these applets can be found in an [appendix](#).

I used the "first edition" of the text in introductory programming courses taught at [Hobart and William Smith Colleges](#) in Fall 1996 and Winter 1998. The second edition has been updated to cover Java 1.1 instead of Java 1.0 and was used in the Fall term of 1998. The course has a weekly lab. Lab worksheets from Fall 1998 and from previous terms are available. (See the [information page](#) for CS124.)

Usage Restrictions

This on-line text can be freely used for non-commercial purposes, as long as its source and author are made clear. For example, you can download a copy and use it on your own computer. You can post it in unmodified form on your own Web server (provided that you do not charge for access). You can print it out for your personal use. Professors who use it in a course can make printed copies and make them available to students for the cost of reproduction.

The text can also be distributed in unmodified form as part of a CD-ROM collection of free and/or shareware materials, provided that the cost of the CD is not more than \$50.

Anyone who wants to use the text for any other purposes that might be considered "commercial" should contact me for permission.

Downloading the Text

This entire text is available for downloading in several formats. The archives, which I haven't yet created as I write this, will probably be between 600 KB and 1 MB in size. You can use the following links to download the archives.

- <http://math.hws.edu/eck/cs124/downloads/javanotes2-fall98.zip>, for Windows 95/98/NT and other platforms.
- <http://math.hws.edu/eck/cs124/downloads/javanotes2-fall98.sit.hqx>, for Macintosh.
- <http://math.hws.edu/eck/cs124/downloads/javanotes2-fall98.tar.Z>, for UNIX.

The "first edition" of the text, which covered Java 1.0 instead of Java 1.1, is also available for download. See the bottom of its index page at <http://math.hws.edu/eck/cs124/notes98.html>.

Why a Free On-line Text?

You might ask, does this really qualify as a textbook? And if so, why is it available for free on-line, instead of as an overpriced hardcover edition?

To answer the first question: Yes, this is meant as a serious textbook. Currently, it is not quite as long as most programming textbooks, but it has plenty of material for a solid one-term course. I think that it is a reasonable choice for a textbook in a college-level programming course -- or I wouldn't be using it in my own courses.

When I started work on the text for the Fall term of 1996, there was really no suitable textbook for introductory programming in Java. I decided to write my own class notes, and it seemed reasonable to put them in HTML format so that I could include working Java applets right on the page. I felt that the result was good enough to publish on the Web, and the response to it has been good. I suppose that I had some idea that I might eventually convert the notes into a hard-copy textbook, but I know from experience that it's a long, hard process to get a textbook into print -- and not a very profitable one unless a lot of people buy the book.

Since then, I've decided that the book really works well in an on-line version. Sometimes, it would be convenient to have a printed version as well, but if I ever do come out with a printed version, it will be a companion to the on-line version, rather than vice versa.

Furthermore, in the meantime, I've become a fan of the Linux operating system and the whole free software movement. (The "free" in this case means "freely distributable" rather than "free of charge.") If we can have free software, why not free textbooks?

Java 1.0 vs. Java 1.1

Java 1.1 introduced a large number of changes to the Java language, and in this second edition of the text, I have made correspondingly large changes. I do **not try to cover both Java 1.0 and 1.1. That is, I almost never say things like, in Java 1.1 you do *this*, but in Java 1.0 you do *that*. And I don't try to point out the features that were unavailable in Java 1.0. It's time to let Java 1.0 fade away...**

However, it is only fairly recently that Web browsers have become available that use Java 1.1. If you read this text with an older browser, most of the applets will just show up as blank white areas. Netscape 4.0.6, released in August 1998, is the first version of Netscape that will run the Java 1.1 applets in these notes. (On the Macintosh, even Netscape 4.0.6 does not support Java 1.1.) Internet Explorer 4.0 also uses Java 1.1, as does Sun Microsystems's browser, HotJava 1.1.4.

Changes from the First Edition

Chapter 1 is almost unchanged, except that I've removed Section 8, which was an explanation of why I decided to use Java instead of C++ in my introductory programming class. I don't think this can any longer be seen as a controversial decision.

In the first edition, Chapters 2 and 3 used a "Console" class that I wrote for doing console-style I/O in programs. I did this because I found standard input and output (System.in and System.out) to be undependable. (The Macintoshes on which I first taught the course did not even implement standard input!) In the second edition, I use a "TextIO" class that simply provides a reasonable interface to the standard

input and output streams. This makes for a smoother exposition, since using the `Console` class forced me to start using objects prematurely.

I've added a new section in Chapter 2 on "the structure of Java programs," in which I try to deal with the confusion that results from having both static and non-static members in classes.

I restructured the material in Chapter 4 extensively, without really adding any important new topics.

The largest changes in the text are in Chapters 5 and 6, which have been completely rewritten to use the Java 1.1 event model. All the applets in the text (except for some of the decorative end-of-chapter applets) have been rewritten to use this event model. I've added sections in Chapter 6 on nested classes and on Frames, and I moved the section on threads and animation from Chapter 6 to Chapter 5. The number of sample applets in Chapters 5 and 6 has been increased substantially.

Chapter 7 contains a new section that briefly introduces some of Java's standard data types, such as `StringBuffer` and `HashTable`. The rest of the chapter is little changed

Chapter 8 has been revised to cover `Reader` and `Writer` streams. These were introduced in Java 1.1 as the recommended way to do character input and output, in place of `InputStream` and `OutputStream`. `InputStream` and `OutputStream` are still used for binary data.

Chapter 9 is essentially unchanged (and might be removed in future editions of this text).

The Future of This Text

I expect that there will be a "third edition" of this text, but not until the second half of the year 2000. I will be on sabbatical for the academic year 1999--2000, so I won't be teaching any courses. However, I do plan to work on this text as one of my sabbatical projects.

It looks like Java is here to stay as an important language. The next version of the language, Java 1.2, will be out before the end of 1998. As far as I know, nothing in Java 1.2 will require major changes in this text. One of the big changes in Java 1.2 will be the inclusion of a new set of GUI components, called "Swing," as an alternative to the AWT components used in Java 1.0 and 1.1. If Swing becomes popular enough to displace the AWT, then I will probably rewrite the text to use Swing instead of the AWT. Most of the other foreseeable changes in Java concern advanced API's that will probably never be more than mentioned in an introductory text

I would like to expand treatment of several topics in the text. In the next edition, Chapter 7 will be broken into at least two chapters. The first chapter will cover arrays, probably with more examples than are now included. The second chapter will include material on linked data structures such as trees, stacks, and queues. It will also include an introduction to recursion. Chapter 8, which in this edition is pretty sketchy, will also be expanded and possibly broken into separate chapters on writing correct and robust programs, using files and streams, and networking. In the longer term, the text might eventually be expanded to include enough material for a two-term introductory programming sequence.

[[Main Index](#)]

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.

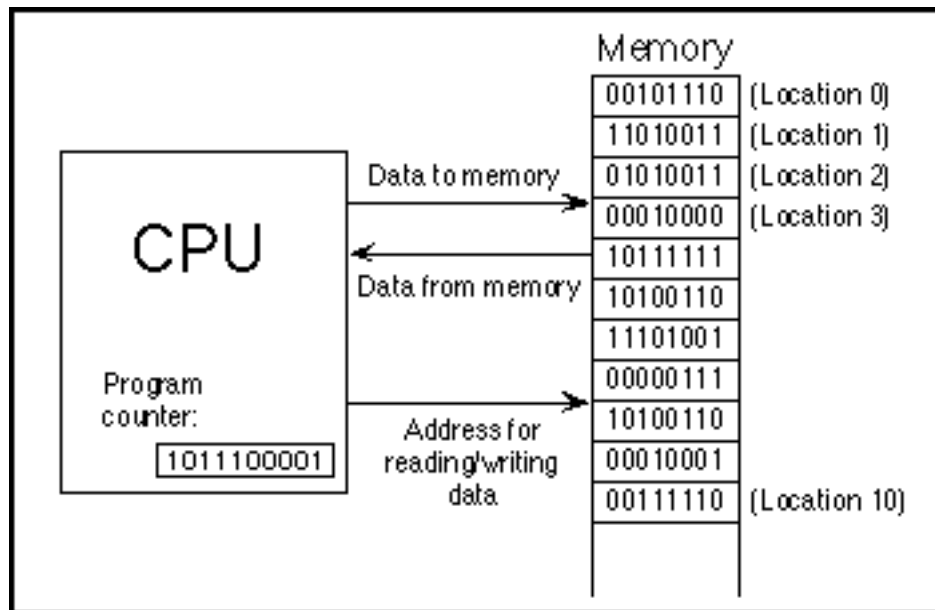
The details of the fetch-and-execute cycle are not terribly important, but there are a few basic things you should know. The CPU contains a few internal **registers**, which are small memory units capable of holding a single number or machine language instruction. The CPU uses one of these registers -- 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

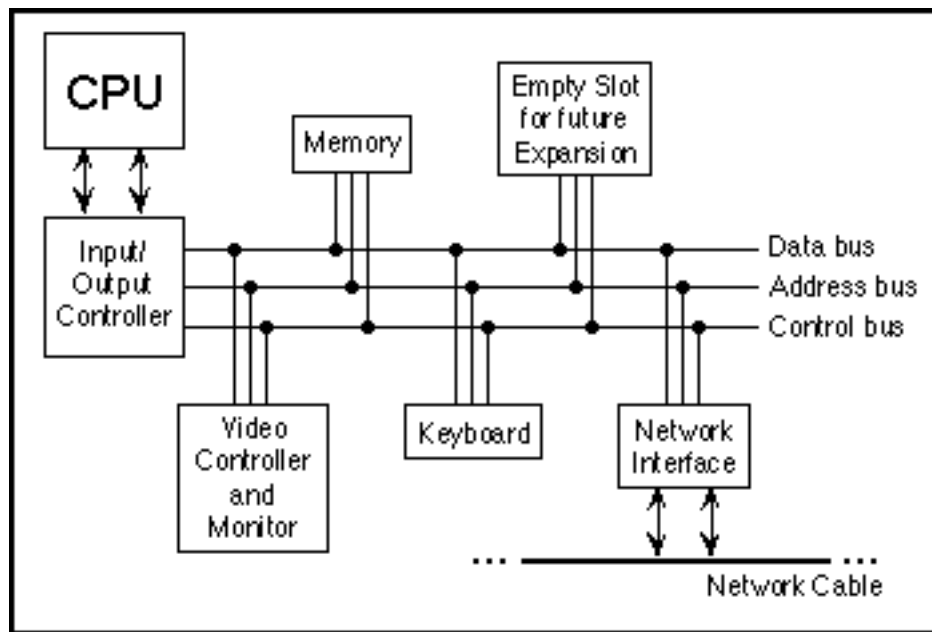
Asynchronous Events: Polling Loops and Interrupts

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 relevant 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.

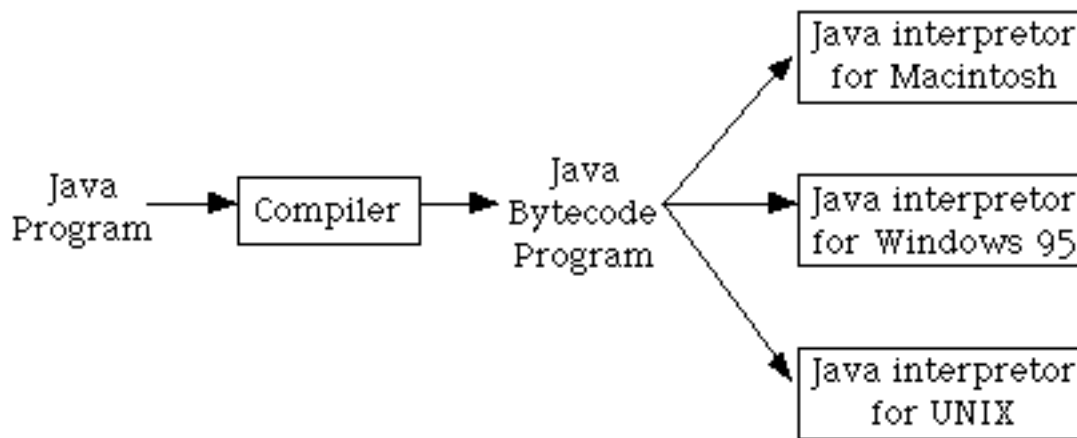
There is an alternative to compiling a high-level language program. Instead of using a compiler, which translates the program all at once, you can use an **interpreter**, which translates it instruction-by-instruction, as necessary. An interpreter is a program that acts much like a CPU, with a kind of 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 95 or 98. (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

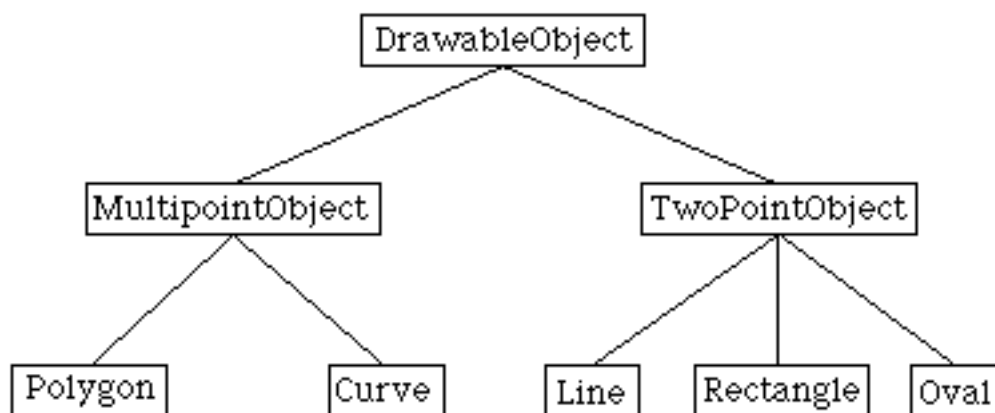
data and subroutines. The point-of-view in OOP is that an object is a kind of self-sufficient entity that has an internal **state** (the data it contains) and that can respond to messages (calls to its subroutines). A mailing list object, for example, has a state consisting of a list of names and addresses. 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

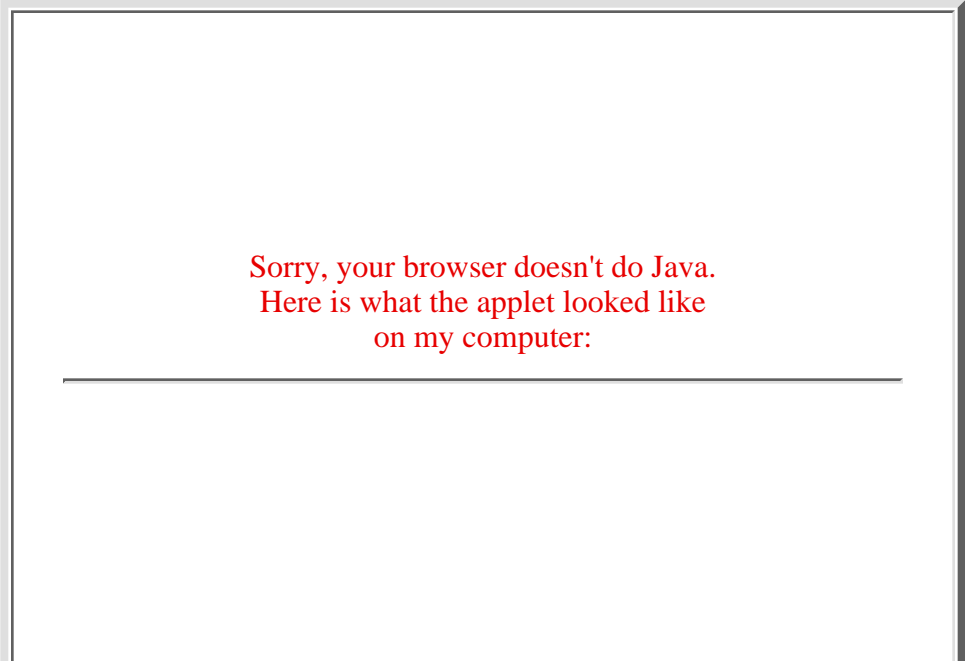
The Modern User Interface

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**.

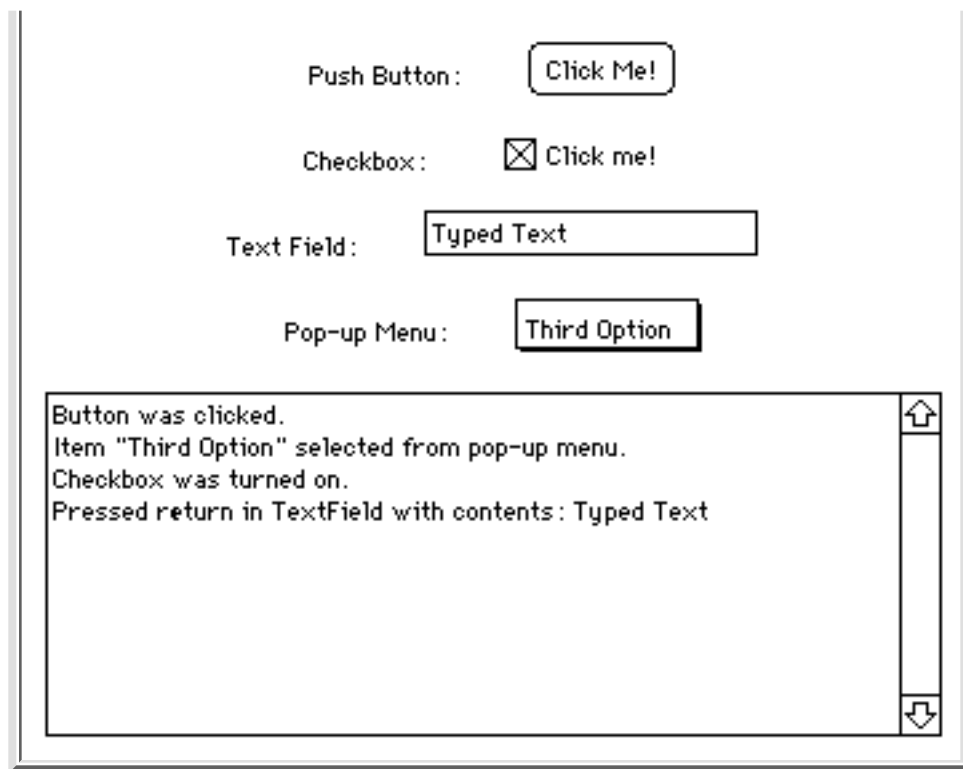
Today, of course, most people interact with computers in a completely different way. They use a **Graphical User Interface**, or GUI. The computer draws interface components on the screen. The components include things like windows, scroll bars, menus, buttons, and icons. Usually, a **mouse** is used to manipulate such components. Assuming that you are reading these notes on a computer, you are no doubt 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 3.1, Windows 98, 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 lower half of the applet is a text area component, that 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 can type in the text field, but you might have to click on it first to activate it:

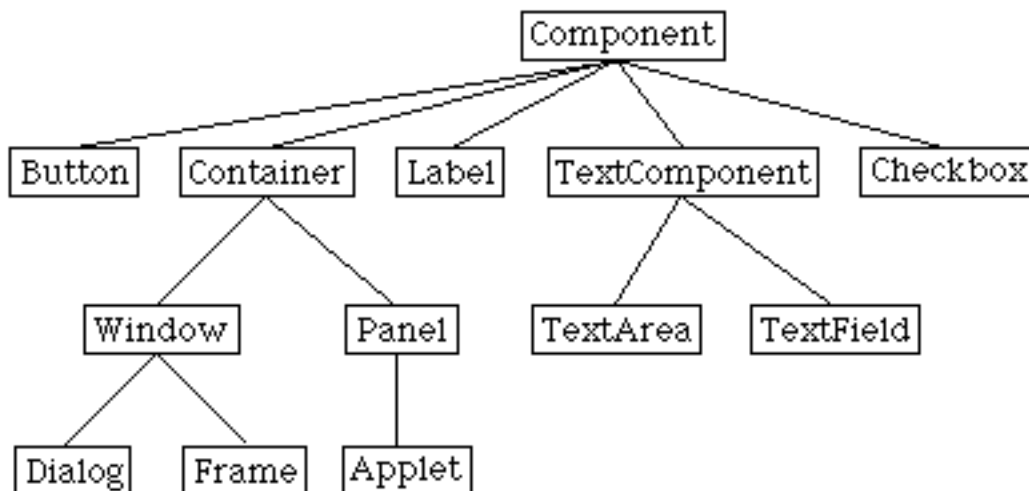


Sorry, your browser doesn't do Java.
Here is what the applet looked like
on my computer:



As you experiment with the other components, you'll find that messages are displayed in the text area. What happens is that when you perform certain actions, such as clicking on a button, you generate "events." For each event, a message is sent to the applet telling it that the event has occurred, and the applet responds according to its program. In fact, the program consists mainly of "event handlers" that tell the applet how to respond to various types of events. In this example, the applet has been programmed to respond to each event by displaying a message in the text area.

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 these classes and their relationships:



Don't worry about the details for now, but try to get some feel about how object-oriented programming and inheritance are used here. Note that all the GUI classes are subclasses, directly or indirectly, of a class called **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](#).

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

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 Telnet, electronic mail, FTP, and the World-Wide Web.

Telnet 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.) Telnet provides only a command-line interface. Essentially, the first computer acts as a terminal for the second. Telnet 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 Telnet, 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 fact should help you understand that FTP is not a program. It 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:

**Sorry, your browser doesn't
support Java.**

[[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 Bill Clinton went fishing. But if I say "Donald Trump wants to be President" I mean that he wants to fill the office, not that he wants to be Bill Clinton.)

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:

Sorry, your browser doesn't
support Java.

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 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 preforms 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.

Sorry, your browser doesn't
support Java.

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 `Strings`:

- `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 you would need to know 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.)

**Sorry, your browser doesn't
support Java.**

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 value of the 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 value. 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.)

**Sorry, your browser doesn't
support Java.**

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 and will change the value of `y` to 6. 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, `|`.) `"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 `x/y` 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, `(x/y > 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

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 could 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 covert 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:

```
* * * * *
```

```
* * * * *
```

```
* *      * *
```

```
* *
```

```
* *      * *
```

```
* *
```

```
* *          * *
```

```
* *
```

```
* *          * *
```

```
* * * * *
```

```
* *          * *
```

```
* *
```

```
* *          * *
```

```
* *
```

```
* *          * *
```

```
* *
```

```
* *      * *
```

```
* *
```

```
* *      * *
```

```
* *
```

```
* *      * *
```

```
* *
```

```
* * * * *
```

```
* * * *
```

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

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 is 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 nickles, 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.getDouble();

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

        /* 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:

Sorry, your browser doesn't

support Java.

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, actually.) 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:

**Sorry, your browser doesn't
support Java.**

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 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.getlnInt();

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

    /* Display the result. */

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

} // end main()

} // end class ComputeAverage

```

Sorry, your browser doesn't
support Java.

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 print 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 $D \% N$ 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

```

Sorry, your browser doesn't
support Java.

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 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

```

**Sorry, your browser doesn't
support Java.**

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");
```

You can check that 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 **statement**'s. If **boolean-expression-1** is `false`, the computer skips **statement-1** and executes the second, nested if statement. To do this, it tests the value of **boolean-expression-2** and uses it to decide between **statement-2** and **statement-3**.

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

```

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

```

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

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

```

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

```

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

```

**Sorry, your browser doesn't
support Java.**

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

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 the standard

class from 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.

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](#).

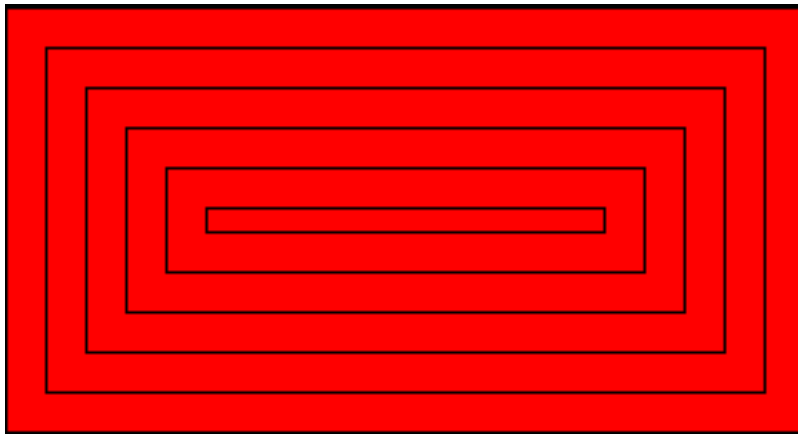
`g.setColor(c)`, is called to set the color that is used for drawing. The parameter, `c` is an object belonging to a class named `Color`, another one of the classes in the `java.awt` package. About a dozen standard colors are available as static member variables in the `Color` class. These standard colors 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, your browser doesn't support Java.
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 therefor 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:

Sorry, your browser doesn't
support Java.

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 `SimpleAnimationApplet`, 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 SimpleAnimationApplet {

    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 `SimpleAnimationApplet` 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 in.

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. My animation framework provides 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 SimpleAnimationApplet {

    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 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
```

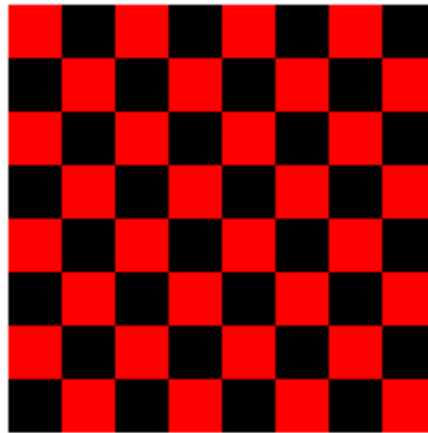
good
idea

(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, `SimpleAnimationApplet`, which was introduced in [Section 7](#). When you run your applet, the compiled class file, `SimpleAnimationApplet.class`, must be in the same directory as your Web-page source file and the compiled class file for your own class. 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 its 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:

Sorry, your browser doesn't
support Java.

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 in 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**. We could write the signature of the subroutine `doTask` 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:

**Sorry, your browser doesn't
support Java.**

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 a return 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 fairly rare 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 testing 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:

**Sorry, your browser doesn't
support Java.**

[[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 3.1 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

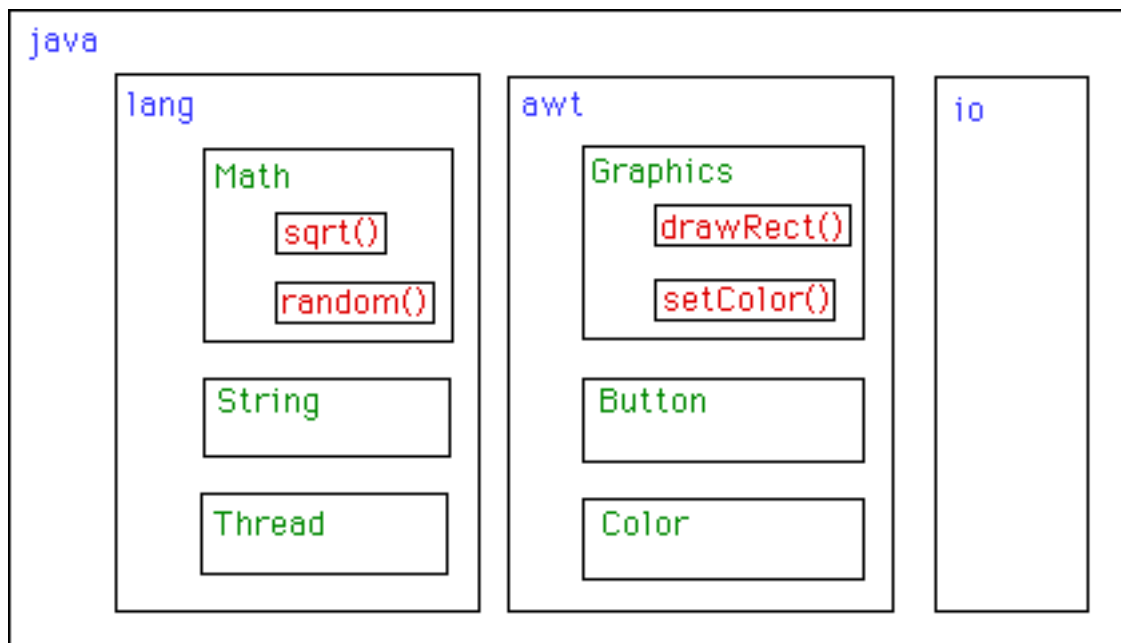
interface that is platform-independent; the implementation varies from one platform to another. A Java system on a particular computer includes implementations of all the standard API routines. A Java program includes only calls to those routines. When the Java interpreter executes a program and encounters 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 as one large package, which is named "java". The java package, in turn, is made up of several other packages, and each of those packages contains a number of classes.

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 graphical user interfaces, such as the 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.)

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, which includes 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.Button 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.Button stopBtn;
```

to declare a variable named `stopBtn` whose type is `java.awt.Button`. 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.Button;
```

at the beginning of your program, before you start writing any class, then you can abbreviate the full name `java.awt.Button` to just the name of the class, `Button`. This would allow you to say just

```
Button stopBtn;
```

to declare the variable `stopBtn`. (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.*;
```

In fact, any Java program that uses a graphical user interface is likely to begin with this line. 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.

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 `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.

Let's work through an example. 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:

`Mosaic.open(rows,cols,w,h);` where `rows`, `cols`, `w`, and `h` are of type `int`. This will open the window with `rows` rows and `cols` columns of rectangles. The size of each little rectangle will be `w` pixels wide by `h` pixels high. Initially, all the rectangles are black. Note that for purposes of referring to a specific rectangle, rows are numbered from 0 to `rows-1`, and columns are numbered from 0 to `cols-1`.

`Mosaic.setColor(row,col,r,g,b);` where all the parameters are of type `int`. This will set the color of the rectangle in row number `row` and column number `col`. Any color can be considered to be a combination of the primary colors red, blue, and green. The parameters `r`, `g`, and `b` are integers in the range from 0 to 255 that specify the red, green, and blue components of the color. The larger the value of `r`, the more red there is in the color. Black has all three color components equal to 0. White has all three color components equal to 255.

`Mosaic.getRed(row,col);` where `row` and `col` are integers specifying one of the rectangles. This is a function that returns a value of type `int`. The returned value is an integer in the range from 0 to 255 that specifies the red component of the color of the specified square. There are also functions `Mosaic.getBlue(row,col);` and

`Mosaic.getGreen(row,col)`; for retrieving the other two color components.

`Mosaic.delay(millis)`; where `millis` is of type `int`. This can be used to insert a time delay in the program (to regulate the speed at which the colors are changed, for example). The parameter `millis` is an integer that gives the number of milliseconds to delay. One thousand milliseconds equal one second.

`Mosaic.isOpen()`; is a function that returns a boolean value. If the mosaic window is open, the value is `true`; otherwise it is `false`. The user can close the window by clicking its closebox. A program can close the window by calling `Mosaic.close()`. If you call `Mosaic.setColor()` when there is no window open, it will have no effect. If you call `Mosaic.getRed()` when there is no window, a value of 0 is returned.

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:

Sorry, your browser doesn't
support Java.

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 that I'll 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. Pseudocode for `fillWithRandomColors()` could be given as:

```
For each row:
    For each column:
        set 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`. The random values must be integers in the range from 0 to 255. A formula for 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.
        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() {
        // fill every square, in each row and column,
        // with 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) {
    // Change the square in row number rowNum and
    // column number colNum to a random color.
    // Random colors in the range 0 to 255 are
    // chosen for the red, green, and blue components
    // of the 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() {
    // Randomly move the disturbance in one of
    // four possible directions: up, down, left, or right;
    // if this moves the disturbance outside the window,
    // then move it to the opposite edge of the window.
    int directionNum; // Randomly set to 0, 1, 2, or 3
                        // to choose the 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:
            currentColumn--;
            if (currentColumn < 0)
                currentColumn = 19;
            break;
    }
} // end of randomMove()

} // end of class RandomMosaicWalk

```

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 in 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 when calling various subroutines in the `Font` class for specifying different styles of text.

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() {
    // Fill every square, in each row and column,
    // with 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) {
    // Change the square in row number rowNum and
    // column number colNum to a random color.
    // Random colors in the range 0 to 255 are
    // chosen for the red, green, and blue components
    // of the 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() {
    // Randomly move the disturbance in one of
    // four possible directions: up, down, left, or right;
    // if this moves the disturbance outside the window,
    // then move it 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:
        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.) However, once the block in which a variable is declared ends, its name does become available for reuse. 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 a pair of dice over and over, 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 slow machines.)

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 50 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 500 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 a console. (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. It also discusses the object-oriented approach to program design.

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: [More Details of Classes](#)
 - [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 variable in the class, `UserData.name` and `UserData.age`. 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 of those objects will have its **own variables** called `name` and `age`. 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 what instance methods objects 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 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`. 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 can be written in Java as `"null"`. You could assign a null reference to 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:

```

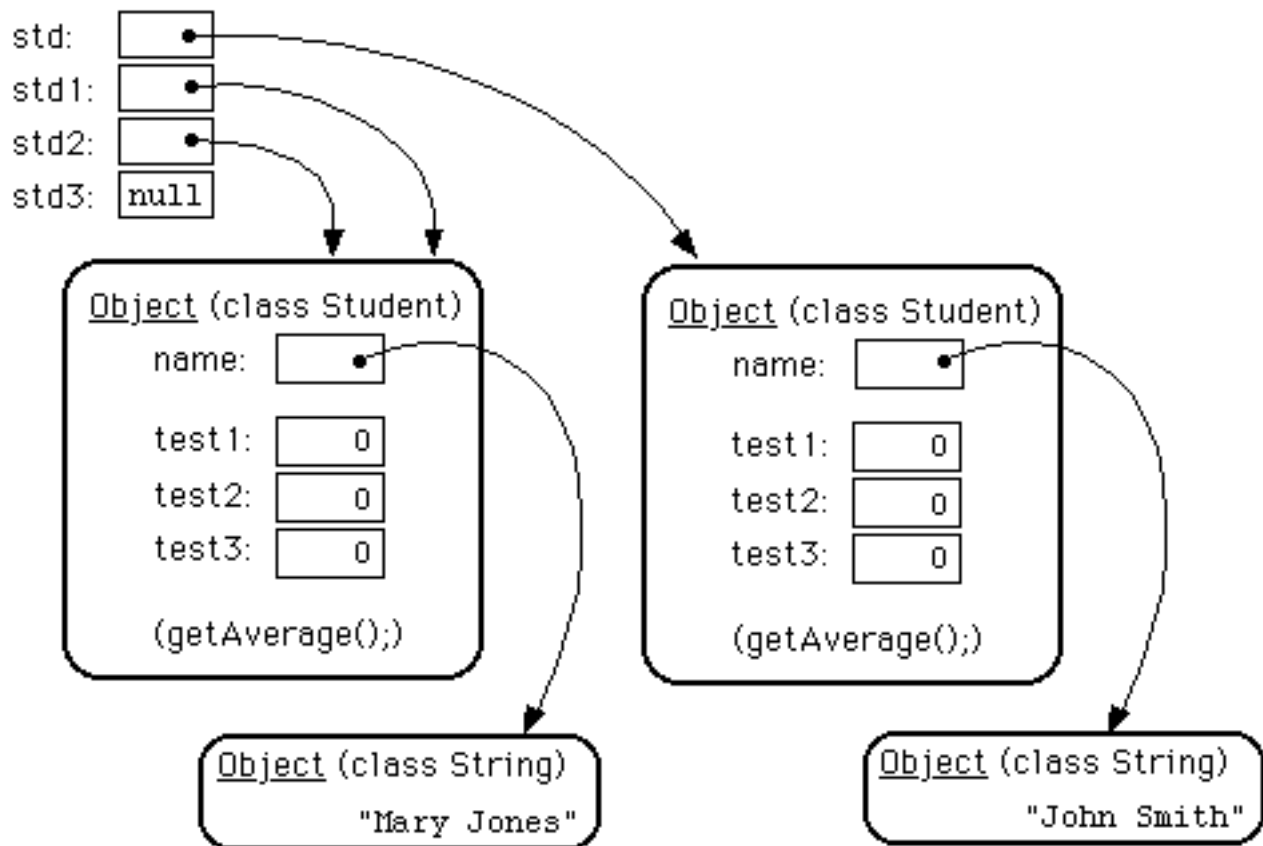
Student std, std1,      // Declare four variables of
    std2, std3;         //   type Student.
std = new Student();    // Create a new object belonging
                        //   to the class Student, and
                        //   store a reference to that
                        //   object in 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` is set to refer to the 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 == std3.test1 && 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 at 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 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:

**Sorry, your browser doesn't
support Java.**

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 change 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 on 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. 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.

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 low
                     // 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:

**Sorry, your browser doesn't
support Java.**

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

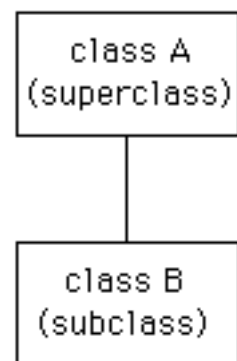
Section 5.4

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.

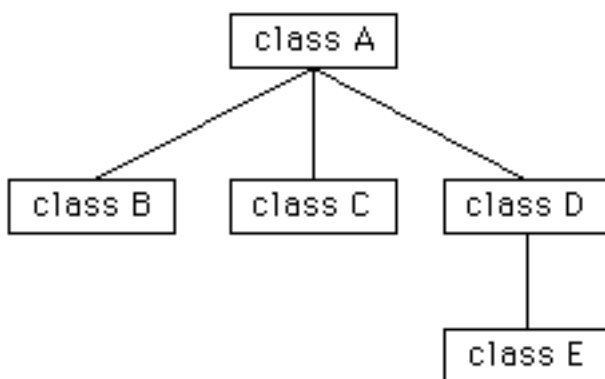
The term **inheritance** refers to the fact that one class can inherit part or all of its structure and behavior from another class. The class that does the inheriting is said to be a **subclass** of the class from which it inherits. If class B is a subclass of class A, we also say that class A is a **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

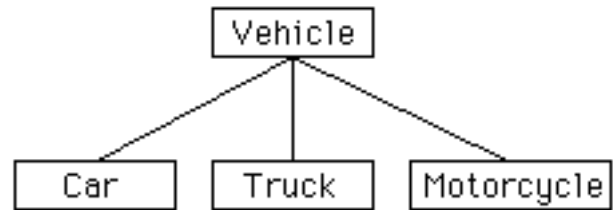
```

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, if `myVehicle` is a variable of type `Vehicle` 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;
```

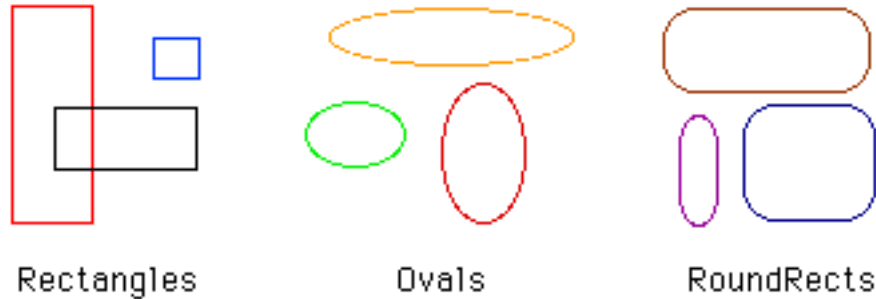
and you could even refer to `((Car)myVehicle).numberOfDoors`. As an example of how this could be used in a program, suppose that you want to print out relevant data about a vehicle. You could say:

```
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
}

```

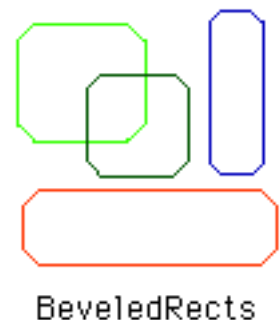
If `oneShape` is a variable of type `Shape`, it could refer to an object of any of the types, `Rectangle`, `Oval`, or `RoundRect`. As a program executes, and the value of `oneShape` changes, it could even refer to objects of different types at different times! Whenever the statement

```
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 computer would say, `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 could 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.Vector`, that represents a list of `Objects`. (The `Vector` 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, `Vector`, instead of the full name, `java.util.Vector`. See [Section 4.5](#) for a discussion of packages and `import`.) The `Vector` class is very convenient, because a `Vector` 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 a `Vector`. Suppose that the `Vector` is named `listOfShapes`. A shape, `oneShape`, can be added to the end of the list by calling the instance method

`"listOfShapes.addElement(oneShape);"`. The shape could be removed from the list with `"listOfShapes.removeElement(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.elementAt(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 `Vector` 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.elementAt(i)` to be a value of type `Shape`:

```
oneShape = (Shape)listOfShapes.elementAt(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.elementAt(i);
    s.redraw();
}
```

It might be worthwhile to look at an applet that actually uses an abstract `Shape` class and a `Vector` to hold a list of shapes:

Sorry, your browser doesn't
support Java.

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 "Choice 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 chapter. (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 a `Vector` is used to hold a list of shapes.

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

Section 5.5

More Details of Classes

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 covers more of those annoying details. You should not necessarily master everything in this section the first time through, but you should read it to be aware of what is possible. (This doesn't apply to the first subsection, below, which you definitely need to master.) For the most part, when I need to use material from this section later in the text, I will explain it again briefly, or I will refer you back to this section.

Extending Existing Classes

The [previous section](#) discussed subclasses, including information about how to program with subclasses in Java. However, that section dealt mostly with the theory. In 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

```

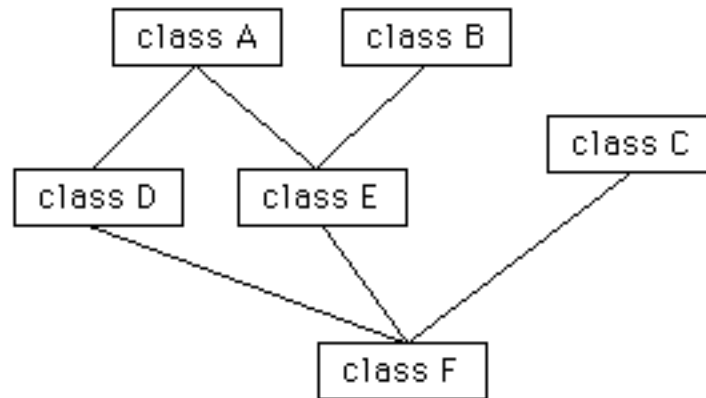
Since `BlackjackHand` is a subclass of `Hand`, an object of type `BlackjackHand` contains all the instance variables and instance methods defined in `Hand`, plus the new instance method `getBlackjackValue()`. For example, if `bHand` is a variable of type `BlackjackHand`, then the following are all legal 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.

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 meaning. An "interface" in Java 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();
}
```

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() {
        . . . // do something -- presumably, draw a line
    }
    . . . // other methods and variables
}
```

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 building other classes. 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();      // calls draw() method from class Line

figure = new FilledCircle(); // Now, figure refers to an object
                             // of class FilledCircle.
figure.draw();             // 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()`, above, is legal because any such class 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.

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 that can be used inside the class where the variable or method is defined. But a class does not actually contain instance variables or methods, only their source code. Actual instance variables and methods are contained in objects. 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 a class definition, and you want to refer to the object that contains the instance method you are writing? Suppose you want to use a full name for an instance variable, because its simple name is hidden by a local variable?

Java provides a special, predefined variable named "`this`" that you can use for these purposes. The variable, `this`, can be used in the source code of an instance method to refer to the object that contains the

method. If `x` is an instance variable, then `this.x` can be used as a full name for that variable. Whenever the computer executes an instance method, it 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 is another common use 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);"`.

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. 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` method should change the values of the dice and redraw the dice to show the new values. The `GraphicalDice` class might look something like this:

```
public class GraphicalDice extends PairOfDice {

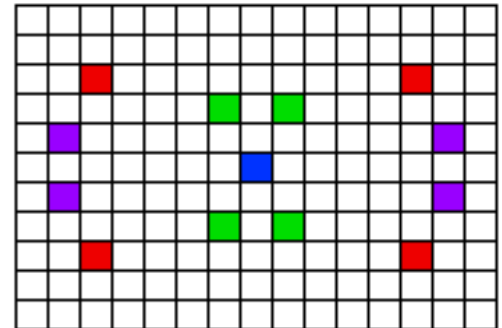
    public void roll() {
        // Roll the dice, and redraw them.
        super.roll(); // Call the roll method from PairOfDice.
        redraw();     // Call a method to draw the dice.
    }

    .
    // 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 this page. 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 at the bottom of this page.

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`, which was introduced in the previous subsection. 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.

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 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.

As an 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.

[[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 are rolled before a given total comes up. It repeats this experiment 10000 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 use, 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](#)
 - Section 2: [HTML Basics and the Web](#)
 - Section 3: [Graphics and the Paint Method](#)
 - Section 4: [Mouse Events](#)
 - Section 5: [Keyboard Events](#)
 - Section 6: [Introduction to Layouts and Components](#)
 - Section 7: [Looking Back: The Java 1.0 Event Model](#)
 - [Programming Exercises](#)
 - [Quiz on this Chapter](#)
-

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

Section 6.1

The Basic Java Applet

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 many applets are meant to be used on Web pages, there are other ways to use them and reasons to do so. 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. You will see a few examples later in this section, and I will discuss graphics in more detail in [Section 3](#).

The `paint()` method of an applet does **not, by the way, draw GUI components such as buttons and text input boxes that the applet might contain**. Such GUI components are objects in their own right, defined by other classes. All component objects, not just applets, have `paint()` methods. Each component is

responsible for drawing itself, in its own `paint()` method. Later, we'll see that many applets do not even define a `paint()` method of their own. Such applets exist only to hold other GUI components, which draw themselves.

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 what this code displays:

**Sorry, but your browser
doesn't do Java.**

If the browser you are using does not support Java, or if you have turned off Java support, then you won't see anything. Otherwise, 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 `HelloWorldApplet` is pretty boring. An applet should do something, either on its own or in response to user actions. As an example, we'll look at an applet in which a friendly greeting changes color whenever the user clicks on a button. This example demonstrates several major aspects of applet programming:

Sorry, but your browser
doesn't support Java.

The button in this applet is an object that belongs to the class `Button` (more properly, `java.awt.Button`). 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. Unlike most objects, applets do not use constructors to do initialization. (More exactly, a constructor would not be a good place to do initialization, since some aspects of the applet, such as its height and width, have not been determined at the time the constructor is called.) Instead, just after an applet object is created, the system calls that object's `init()` method, which has the form

```
public void init() { . . . }
```

This method can do the task usually performed by a constructor, that is, to initialize the applet's instance variables. The `init()` method is also the place where other components, such as buttons, are added to the applet.

Once it has been added to the applet, a `Button` object mostly takes care of itself. In particular, it draws itself. 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? The color of the message in the applet should change. It is tempting to think that the `actionPerformed()` method should simply redraw the message in a new color. Unfortunately, there is a problem.

The applet's `paint()` method is called by the system as soon as the applet appears on the screen. But it can also be called at other times. In fact, it is called whenever the contents of the applet need to be redrawn. This might happen if the applet is covered up by another window and is then uncovered. It can happen when you scroll the window of a browser, and the applet scrolls into view. And, it is especially important to note, the applet's `paint()` method can be called because the program makes a request for the applet to be redrawn. Such requests are made by calling a method named `repaint()`.

The `paint()` method can be called over and over, and each time it's called, it must be able to draw the correct picture in the applet. In the applet we are writing, that means drawing the message in the correct color. Suppose our `actionPerformed()` method simply draws "Hello World" in a new color. The `paint()` method must be able to redraw the message in the same color. How will the `paint()` method know which color to use? The only way an object "knows" anything is by having data stored in its instance variables. Our applet needs an instance variable to store information about the color of the message. The `actionPerformed()` method sets the value of that instance variable. The `paint()` method checks the value of the variable to decide which color to use for the message. In fact, the `actionPerformed` method doesn't have to do any drawing at all! It just sets the instance variable and calls `repaint()`. In response to the `repaint()` call, the system will call the `paint()` routine, and that is where the actual drawing happens.

Given all this, you can understand a lot of what goes on in the source code for our colored Hello World applet. This example shows several aspects of applet programming: An `init()` method sets up the applet and adds components; a `paint()` method draws the applet based on data stored in instance variables; and an event-handling method says what happens in response to certain user actions. I've included comments in the source code to explain other aspects of the programming, which will be covered in full later in this chapter. With this help, You should be able to follow what is going on

```
// An applet that says "Hello World" in a big bold font,
// with a button to change the color of the message.

import java.awt.*;           // Defines basic classes for GUI programming.
import java.awt.event.*;     // Defines classes for working with events.
import java.applet.*;       // Defines the applet class.

public class ColoredHelloWorldApplet
    extends Applet implements ActionListener {

    // Defines a subclass of Applet.  The "implements ActionListener"
    // part says that objects of type ColoredHelloApplet are
    // capable of listening for ActionEvents.  This is necessary
    // if the applet is to respond to events from the button.

    int colorNum;           // Keeps track of which color is displayed;
                           //      1 for red, 2 for blue, 3 for green.

    Font textFont;         // The font in which the message is displayed.
                           // A font object represent a certain size and
                           // style of text drawn on the screen.

    public void init() {

        // This routine is called by the system to initialize
        // the applet.  It sets up the font and initial colors
        // the applet.  It adds a button to the applet for
        // changing the message color.

        setBackground(Color.lightGray);
        // The applet is filled with the background color before
        // the paint method is called.  The button and the message
        // in this applet will appear on a light gray background.

        colorNum = 1;      // The color of the message is set to red.

        textFont = new Font("Serif",Font.BOLD,24);
        // Create a font object representing a big, bold font.

        Button btnn = new Button("Change Color");
        // Create a new button.  "ChangeColor" is the text
        // displayed on the button.

        btnn.addActionListener(this);
        // Set up btnn 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.
```

```
        add(btn);    // Add the button to the applet, so that it
                     // will appear on the screen.

    } // end init()

    public void paint(Graphics g) {

        // This routine is called by the system whenever the content
        // of the applet needs to be drawn or redrawn.  It displays
        // the message "Hello World" in the proper color and font.

        switch (colorNum) {           // Set the color.
            case 1:
                g.setColor(Color.red);
                break;
            case 2:
                g.setColor(Color.blue);
                break;
            case 3:
                g.setColor(Color.green);
                break;
        }

        g.setFont(textFont);          // Set the font.

        g.drawString("Hello World!", 20,70);    // Draw the message.

    } // end paint()

    public void actionPerformed(ActionEvent evt) {

        // This routine is called by the system when the user clicks
        // on the button.  The response is to change the colorNum
        // which determines the color of the message, and to call
        // repaint() to see that the applet is redrawn with the
        // new color.

        if (colorNum == 1)            // Change colorNum.
            colorNum = 2;
        else if (colorNum == 2)
            colorNum = 3;
        else
            colorNum = 1;

        repaint(); // Tell system that this applet needs to be redrawn

    } // end init()

} // end class ColoredHelloWorldApplet
```

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 "nearby" 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).

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 1.1.

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 the Paint Method

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. Note that all the classes mentioned in this section are defined in the package `java.awt`.

For most of this chapter, we'll be drawing directly on applets, usually in the applets' `paint()` methods. In [Section 6](#), we'll encounter another class of GUI component, `Canvas`, that exists only to be drawn on. In many cases, a program does all its drawing in a `Canvas` which is just one of several components contained in the applet. The `Canvas` class, the `Applet` class, and in fact all of the classes that represent GUI components are subclasses of another class, named `Component`. The `Component` class represents the general idea of a Graphical User Interface component that can appear on the screen. Many of the methods used in applets, including `paint()` and `repaint()`, are actually inherited from `Component`. Most of what I will say about graphics applies to any `Component`, not just to `Applets` and `Canoases`. Whenever I talk about GUI components, I am referring to all objects that belong to subclasses of the `Component` class.

To do any drawing at all in Java, you need a **graphics context**. A graphics context is an object belonging to the class, `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 one of Java's GUI components, such as an applet. The `Graphics` class is an abstract class, which means that it is impossible to create a graphics context directly, with a constructor. There are two ways to get a graphics context for drawing on a component: First of all, of course, when a component's `paint()` method is called, the parameter to that method is a graphics context for drawing on the component. Second, to make it possible to draw on a component from outside its `paint()` method, every component has an instance method called `getGraphics()`. This method is a function that returns a graphics context for the component. (The official line is that **all drawing in a component should be done in that component's `paint()` method**, but I have found that this is not always practical and does not always give acceptable performance. Anyway, if the people who designed Java really meant it, they wouldn't have made the `getGraphics()` method public.)

The instance method, `getGraphics()`, is defined in the `Component` class. It returns a graphics context that can be used for drawing to a particular component. That is, if `comp` is any component object and if `g` is a variable of type `Graphics`, then you can say

```
g = comp.getGraphics();
```

After this assignment, `g` can be used for drawing to the rectangular area of the screen that represents the component, `comp`. When you are writing your own applet or other component class, you need to call the (inherited) `getGraphics()` method in the same class. So, you would say simply "`g = getGraphics()`". This gives you a graphics context for drawing in the component you are writing.

If `g` is a graphics context that you've obtained with the `getGraphics()` methods, it is a good idea to call the method `g.dispose()` after you have finished using it. This method frees any system resources that are used by the graphics context. This is a good idea because on many systems, such resources are limited. However, you should never call `dispose()` for the graphics context provided in the `paint()` method. And you should never try to use a graphics context that has been disposed.

Paint, Repaint, and Update

Most components do, in fact, do all drawing operations in their `paint()` methods. The `paint()` method should be smart enough to correctly redraw the component at any time, using data stored in instance variables that record the state of the component. If in the middle of some other method you realize that the appearance of the component should change, then you should change the values of the instance variables and call the component's `repaint()` method. This tells the system that it should redraw the component as soon as it gets a chance (by calling the component's `paint()` method). This approach is satisfactory in most cases. The alternative approach -- drawing directly to the applet with a graphics context obtained through `getGraphics()` -- should be used only when `repaint()` doesn't give satisfactory results.

Now, as it happens, the system does not actually call the `paint()` method directly. There is another method called `update()` which is the one actually called directly by the system. The built-in update procedure first fills in the entire component with a background color. Then it calls the `paint()` method. The `paint()` method draws on a rectangular area that has already been filled with the background color. Usually, this is the right thing to do. However, if the paint method itself fills in the entire rectangle, so that none of the background color is visible, then filling in the background was a wasted step. In that case, you can override `update()` to read simply:

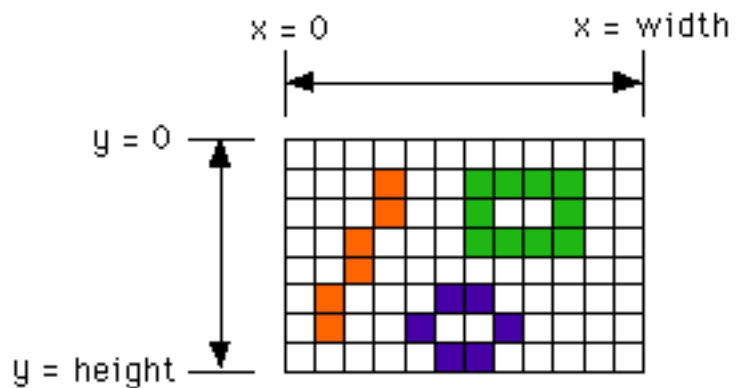
```
public void update(Graphics g) {
    paint(g); // Don't fill with background color; just call paint.
}
```

It is possible to set the background color of a component, using the component's `setBackground()` instance method that I will discuss below.

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, `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 `paint()` method that looks like:

```
public void paint(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, `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 their numbers or type of parameters 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.

Sorry, your browser doesn't
support Java.

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)`. 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**. When the component is filled by the `update()` method, it is filled with the background color. 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. These methods are commonly used to set the foreground and background colors of an applet in the applet's `init()` method.

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 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 `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 tails 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` parameter 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 an applet. 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:

**Sorry, your browser doesn't
support Java.**

The applet does have a problem. When the `paint()` 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 `paint()` 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 is to compute the contents of the picture elsewhere, outside the `paint()` method. Information about the picture should be stored in instance variables, and the `paint()` method should use that information to draw the picture. If `paint()` is called twice before the data is

recomputed, it should draw the same picture twice. 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 hold represents different sizes and styles of text. These variables are initialized in the applet's `init()` method. I also use the `init()` method to set the background color of the applet to black.

The `paint` method 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.

Here is the complete source code:

```

/*
   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.
*/

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

public class RandomStrings extends Applet {

    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.

    public void init() {

        message = getParameter("message");    // Look for message in an
                                                // applet param named
                                                // "message".

        if (message == null)    // If no message is found, use "Java!".
            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);
    }

```

```

        setBackground(Color.black);

    } // end init()

    public void paint(Graphics g) {

        int width = getSize().width;    // Get applet's width and height.
        int height = getSize().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 be a bright, saturated color,
            // with a 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 paint()

} // 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 a button on the 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 5.5](#). (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 Applet 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 `addActionListener()` method is an instance method in the class, `Component`. In particular, since an applet is a component, every applet has an `addMouseListener()`, and so it is possible to set up a listener to respond to clicks on the applet.

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 the class for the subroutines from that interface;
4. Register the listener object with the applet or other component.

Any object can act as a listener, if it implements the appropriate interface. It is considered good form to define new classes just for listening. Unfortunately, doing this effectively requires some rather advanced techniques. (See [Section 7.6](#).) We'll use another strategy, which works well for small projects: Since an applet is itself an object, we'll let the applet itself listen for events. This means that the applet class will be declared to implement any necessary listener interfaces, and it will include the necessary methods to respond to the events.

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. In most cases, you should

define `mousePressed` instead of `mouseClicked`. The other two methods are called when the mouse cursor enters or leaves the component. If you wanted 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:

Sorry, your browser doesn't
support Java.

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 the applet class implements the `MouseListener` interface by saying:

```
class RandomStrings extends Applet implements MouseListener { ...
```

Third, define the five methods of the `MouseListener` interface. Only `mousePressed` will do anything, and all it has to do is call `repaint()` to force the applet to be redrawn. The following methods are added to the class definition:

```
    public void mousePressed(MouseEvent evt) {
        // When user presses the mouse, tell the system to
        // call the applet's paint() method.
        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. This should be done when the applet is initialized, that is, in the applet's `init()` method. This line should be added to the `init()` method:

```
addMouseListener(this);
```

This might need some explanation. We want to listen for mouse events on the applet itself, so we call the applet's own `addMouseListener` method. The parameter to this method is the object that will be doing the listening. In this case, the object is, again, the applet itself. `"this"` is a special variable that refers to the applet. (See [Section 5.5](#).) So, we are telling the applet to listen for mouse events on itself. All this means, effectively, is that our `mousePressed` method will be called when the user clicks on the applet.

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.*;
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 itself.
        super.init();
        addMouseListener(this);
    }

    public void mousePressed(MouseEvent evt) {
        // When user presses the mouse, tell the system to
        // call the applet's paint() method.
        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) { }

} // 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 component's coordinate system, 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. 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. (I draw black outlines around the ovals and rectangles so that they will look nice when they overlap.)

Sorry, your browser doesn't
support Java.

The source code for this applet follows. 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). This applet has no `paint()` method. More properly speaking, it has the inherited `paint()` method that doesn't draw anything. So, when the applet is repainted, it is simply filled with the background color. We still have the problem that we are not storing information about what is drawn on the applet. So if the applet is covered up and uncovered, the contents of the applet are erased.

You should pay attention to how the graphics context, `g`, is used in the `mousePressed` routine. Since I am drawing on the applet from outside its `paint()` method, I need to obtain a graphics context by saying "`g = getGraphics()`". I use `g` to draw an oval or rectangle on the applet, centered on the point where the user clicked. Finally, the graphics context is disposed by calling `g.dispose()`.

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

public class SimpleStamper extends Applet implements MouseListener {

    public void init() {
        // When the applet is created, set its background color
        // to black, and register the applet to listen to mouse
        // events on itself.
        setBackground(Color.black);
        addMouseListener(this);
    }

    public void mousePressed(MouseEvent evt) {
        // This method will be called when the user clicks the
        // mouse on the applet.

        if ( evt.isShiftDown() ) {
            // The user was holding down the Shift key.  Just
            // repaint the applet, which will fill it with its
            // background color, black.
            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 on the applet.

        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 - 25, y - 15, 60, 30 );
        g.setColor(Color.black);
        g.drawOval( x - 25, y - 15, 60, 30 );
    }
    else {
        // Draw a red rectangle centered at the point (x,y).
        g.setColor(Color.red);
        g.fillRect( x - 25, y - 15, 60, 30 );
        g.setColor(Color.black);
        g.drawRect( x - 25, y - 15, 60, 30 );
    }

    g.dispose(); // We are finished with the graphics context,
                // so dispose of it.

} // end mousePressed()

// 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) { }

} // 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 the applet, then the definition of the applet class has the form:

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

public class Mouser extends Applet
    implements MouseListener, MouseMotionListener {

    public void init() {      // set up the applet
        addMouseListener(this);
        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. (The applet flickers annoyingly as the mouse is moved. This is something that can be fixed with a technique called "double buffering" that will be covered in [Section 7.1](#).)

Sorry, your browser doesn't
support Java.

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();
    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.)

Sorry, your browser doesn't
support Java.

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.

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 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 is 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 just draws a line from the previous location of the mouse to its current location.

[[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 in Java.

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. Unfortunately, a component that requires this treatment on one platform might not require it on another platform. In Sun Microsystem's implementation of Java, for example, applet objects must be treated in this way. So if you create a subclass of `Applet` that is supposed to be able to respond to keyboard events, you should be sure to set up a mouse listener for your class, and call `requestFocus()` in the `mousePressed()` method. If you don't do this, your applet might work on some versions of Java, but on others it will fail because it never receives the input focus

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, typing the character 'R', 'G', 'B', or 'K' 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. 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.

Sorry, your browser doesn't
support Java.

In Java, keyboard event objects belong to a class called `KeyEvent`. An object that needs to listen for `KeyEvents` must implement the interface, `KeyListener`. Furthermore, the object must be registered with a component by calling the component's `addKeyListener()` method. When an applet is to listen

for keyboard events on itself, the registration is done with the command `"addKeyListener(this);"` in the applet's `init()` method. 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.

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 a `keyPressed` event. 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, 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 repaint() to redraw the whole 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.

    int key = evt.getKeyCode(); // Keyboard code for the pressed key.

    if (key == KeyEvent.VK_LEFT) {
        squareLeft -= 8;
        if (squareLeft < 3)
            squareLeft = 3;
        repaint();
    }
    else if (key == KeyEvent.VK_RIGHT) {
        squareLeft += 8;
        if (squareLeft > getSize().width - 3 - SQUARE_SIZE)
            squareLeft = getSize().width - 3 - SQUARE_SIZE;
        repaint();
    }
    else if (key == KeyEvent.VK_UP) {
        squareTop -= 8;
        if (squareTop < 3)
            squareTop = 3;
        repaint();
    }
    else if (key == KeyEvent.VK_DOWN) {
        squareTop += 8;
        if (squareTop > getSize().height - 3 - SQUARE_SIZE)
            squareTop = getSize().height - 3 - SQUARE_SIZE;
        repaint();
    }
} // end keyPressed()
```

Color changes -- which happen when the user types the characters 'R', 'G', 'B', and 'K', or the lower case equivalents -- are handled in the keyTyped method. I won't include it here, since it is so similar to the keyPressed method. Finally, to complete the KeyListener interface, the keyReleased method must be defined. In the sample 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. The applet's `paint()` method looks at the value of `focussed` to decide what color the border of the applet should be. The value of the variable is set in the `focusGained()` and `focusLost()` methods. These methods call `repaint()` so that the applet will be redrawn with the correct border color. The method definitions are very simple:

```
public void focusGained(FocusEvent evt) {
    // The applet now has the input focus.
    focussed = true;
    repaint(); // redraw with cyan border
}

public void focusLost(FocusEvent evt) {
    // The applet has now lost the input focus.
    focussed = false;
    repaint(); // redraw with gray border
}
```

The other aspect of handling focus is to make sure that the applet requests the focus when the user clicks on it. To do this, the applet implements the `MouseListener` interface and listens for mouse events on itself. It defines a `mousePressed` routine that asks for the input focus:

```
public void mousePressed(MouseEvent evt) {
    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 Applet
    implements KeyListener, FocusListener, MouseListener
```

The `init()` method registers the applet to listen for all three types of events. To do this, the `init()` method includes the lines

```
addFocusListener(this);
addKeyListener(this);
addMouseListener(this);
```

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 the 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 `paint()` 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 `paint()` 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 `KeyboardAnimationApplet`. (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 `KeyboardAnimationApplet` 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 [KeyboardAnimationApplet.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 object 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.

Sorry, your browser doesn't
support Java.

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. When no explosion is happening, the value of

`explosionFrameNumber` is zero.

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. 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 `KeyboardAnimationApplet` 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()`. 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 positioned 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. But several other things can happen. If the ball 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 drag events. Furthermore, on many platforms, a button can receive the input 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 `Button` class. A `Button` object draws itself and processes mouse, mouse dragging, keyboard, and focus events on its own. You only hear from the `Button` when the user triggers it by clicking on it or pressing the space bar while the button has the input focus. When this happens, the `Button` object creates an event object belonging to the class `ActionEvent`. That event 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 is an example of a container. An independent window is another type of container. Java also has a class of container called `Panel`. Because a `Panel` object is a container, it can hold other components. But a `Panel` is itself meant to be placed inside another container. This allows complex nesting of components. `Panels` 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. As a first example, here is a new version of the `ColoredHelloWorldApplet` from the [Section 1](#). Click the buttons to change the color of the message:

Sorry, but your browser
doesn't support Java.

It is possible to draw directly on an applet, as I've done previously in this chapter. However, it is not a good idea to do so when the applet contains components that will be laid out by a layout manager. The reason is that it's hard to be sure exactly where the components will be placed by the layout manager and how big they will be (If you knew that, you wouldn't be using the layout manager! It's supposed to make such computations, so you don't have to.) A better idea is to use a special-purpose component to display the drawing. This drawing component or "canvas" will be just one of the components contained in the applet. The white rectangle in the above applet, where the "Hello World" message is displayed, is an example of such a component. This component is a member of a class `ColoredHelloWorldCanvas`, which I have defined as a subclass of the standard class, `java.awt.Canvas`. The `Canvas` class exists precisely for creating such drawing areas. An object that belongs to the `Canvas` class itself is nothing but a patch of color. To create a canvas with content, you have to define a subclass of `Canvas` and write a `paint()` method for your subclass to draw the content you want.

When you use a canvas in this way, it's a good idea to put all the information necessary to do the drawing in the canvas object, rather than in the main applet object. The original `ColoredHelloWorldApplet` used an instance variable, `textColor`, to keep track of the color of the displayed message. In the new version, the `textColor` variable is moved to the `ColoredHelloWorldCanvas` class. This class also contains a method, `setTextColor()`, which can be called to tell the canvas to change the color of the message. When the user clicks one of the buttons, the applet responds by calling this method. This is good object-oriented program design: The `ColoredHelloWorldCanvas` class is responsible for displaying a colored greeting, so it should contain all the data and behaviors associated with its role. On the other hand, this class doesn't need to know anything about buttons, layouts, and events. Those are the job of the main applet class. This separation of responsibility helps reduce the overall complexity of the program.

So, here's the `ColoredHelloWorldCanvas` class:

```
class ColoredHelloWorldCanvas extends Canvas {

    // A canvas that displays the message "Hello World" on
    // a white background in a big, bold font. A method is
    // provided for changing the color of the message.

    Color textColor; // Color in which "Hello World" is displayed.
    Font textFont;   // The font in which the message is displayed.

    ColoredHelloWorldCanvas() {
        // Constructor.
        setBackground(Color.white);
        textColor = Color.red;
        textFont = new Font("Serif",Font.BOLD,24);
    }

    public void paint(Graphics g) {
        // Show the message in the set color and font.
        g.setColor(textColor);
        g.setFont(textFont);
        g.drawString("Hello World!", 20,40);
    }

    void setTextColor(Color color) {
        // Set the text color and tell the system to repaint
        // the canvas so the message will be in the new color.
        textColor = color;
        repaint();
    }
}
```

```
    } // end class ColoredHelloWorldCanvas
```

You should be able to understand this. Just keep in mind that all this applies to the canvas, not to the whole applet. When the constructor calls the method `setBackground()`, it's only the background of the canvas that is set to white. The background color of the applet is not affected. Remember that every component has its own background color, foreground color, and font. The `setTextColor()` method is called by the applet when it wants to change the color of the displayed message. Note that `setTextColor()` calls `repaint()`. Since this `repaint()` method is in the `ColoredHelloWorldCanvas` class, it causes just the canvas to be repainted, not the whole applet. Every component has its own `paint()` and `repaint()` methods, and every component is responsible for drawing itself. This is another example of the way responsibilities are distributed in an object-oriented system.

The main applet class, `ColoredHelloWorldApplet2`, is responsible for managing all the components in the applet and the events they generate. The components include the drawing canvas and three buttons. These components are created and added to the applet in the applet's `init()` method. The applet will listen for `ActionEvents` from the buttons, so the applet class implements the interface, `ActionListener`. The applet's `init()` method sets up the applet to listen for `ActionEvents` from each button. It does this by calling the button's `addActionListener` method.

The buttons are not contained directly in the applet. Instead, they are added to a `Panel`, and that panel is added to the applet. The `Panel` is the gray strip across the bottom of the applet. Every container object, including applets and panels, include several `add()` methods that are used to add components to the container. For example, if `btn` is a button, and `panel` is a container, then the command `"panel.add(btn);"` adds the button to the panel. This means that the button will appear in the panel. Exactly where it shows up depends on the panel's layout manager.

Once the canvas and panel have been created, it's time to lay out the applet as a whole. This is done by the last three lines of the `init()` method. I use a "BorderLayout," as the layout manager for the applet. A `BorderLayout` displays one big component in the "Center" of the applet and, optionally, up to four other components along the edges of the applet in the "North", "South", "East", and "West" positions. The component in the center gets any space that is left over after the other components are drawn. In this example, the canvas is added in the "Center" position, with the button bar below it, to the "South". When a component is added to a container that uses a `BorderLayout`, the `add()` method has to specify which of the five possible positions should be used for the component:

Here is the complete `init()` method from the applet class:

```
public void init() {

    // This routine is called by the system to initialize the
    // applet. It creates the canvas and lays out the applet
    // to consist of a bar of control buttons below the canvas.

    setBackground(Color.lightGray);

    canvas = new ColoredHelloWorldCanvas();

    Panel buttonBar = new Panel(); // panel to hold control buttons

    Button redBtn = new Button("Red"); // Create buttons, add them
    redBtn.addActionListener(this);    // to the button bar. The
    buttonBar.add(redBtn);              // parameter to the Button
                                      // constructor is the text
                                      // that appears on Button.
```

```

        Button greenBttn = new Button("Green");
        greenBttn.addActionListener(this);
        buttonBar.add(greenBttn);

        Button blueBttn = new Button("Blue");
        blueBttn.addActionListener(this);
        buttonBar.add(blueBttn);

        setLayout(new BorderLayout(3,3)); // Set layout for applet.
        add(buttonBar, BorderLayout.SOUTH); // Put panel at the bottom.
        add(canvas, BorderLayout.CENTER); // Canvas will fill any
                                          // remaining space.
    } // end init()

```

You might not understand this completely just yet, but if you want to write an applet using a similar layout, you can follow the pattern in this sample `init()` method.

In order to handle the `ActionEvents` from the buttons, the applet must define an `actionPerformed()` method. This is the only method specified by the `ActionListener` interface. This method has a parameter, `evt`, of type `ActionEvent`. This parameter can be used to find out which button is responsible for the action event. The function `evt.getActionCommand()` returns a `String` that gives the text that the button displays. The `actionPerformed()` method in the sample applet checks the value returned by `evt.getActionCommand()` and sets the text color of the canvas to the appropriate value. (Remember that `canvas` is an object belonging to the class `ColoredHelloWorldCanvas`, which is shown above.)

```

    public void actionPerformed(ActionEvent evt) {

        String command = evt.getActionCommand();

        if (command.equals("Red"))
            canvas.setTextColor(Color.red);
        else if (command.equals("Green"))
            canvas.setTextColor(Color.green);
        else if (command.equals("Blue"))
            canvas.setTextColor(Color.blue);

    } // end init()

```

There is just one more method in the applet, and it requires a little explanation:

```

    public Insets getInsets() {
        return new Insets(3,3,3,3);
    }

```

This method is called by the layout manager to decide how much space to leave between the edges of the applet and the components that the applet contains. The background color of the applet will show though in this border. The object of type `Insets` that is returned by this method specifies a 3-pixel border along each edge of the applet. There is also, by the way, a 3-pixel boundary between components in the applet. This is specified in the constructor of the layout manager, "`new BorderLayout(3,3)`", in the applet's `init()` method.

You can see how all this is put together in the source code for the applet, [ColoredHelloWorldApplet2.java](#). (Note that the source code for the canvas class is in a separate file, [ColoredHelloWorldCanvas.java](#), and you need both classes in order to use the applet.)

As 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.

Sorry, but your browser
doesn't support Java.

The overall form of this applet is the same as that of `ColoredHelloWorldApplet2`: It has three buttons in a panel at the bottom of the applet and a large canvas for drawing. Of course, in this case the canvas does more interesting things. The canvas class includes all the programming for the game. Since that was true, I decided to let the canvas class act as `ActionListener` and respond to the buttons directly. The applet class, which just sets everything up, is fairly simple:

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

public class HighLowGUI extends Applet {

    public void init() {

        // The init() method lays out the applet using a BorderLayout.
        // 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) );
        setLayout( new BorderLayout(3,3) );

        HighLowCanvas board = new HighLowCanvas();
        add(board, BorderLayout.CENTER); // Add canvas to the applet.

        Panel buttonPanel = new Panel();
        buttonPanel.setBackground( new Color(220,200,180) );
        add(buttonPanel, BorderLayout.SOUTH); // Add panel to applet.

        Button higher = new Button( "Higher" );
        higher.addActionListener(board); // BOARD LISTENS, NOT APPLET!
        higher.setBackground(Color.lightGray);
        buttonPanel.add(higher);

        Button lower = new Button( "Lower" );
        lower.addActionListener(board);
        lower.setBackground(Color.lightGray);
        buttonPanel.add(lower);

        Button newGame = new Button( "New Game" );
        newGame.addActionListener(board);
        newGame.setBackground(Color.lightGray);
        buttonPanel.add(newGame);
    }
}
```

```

    } // end init()

    public Insets getInsets() {
        return new Insets(3,3,3,3);
    }

} // end class HighLowGUI

```

In programming the canvas 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 that are displayed. The cards are stored in an object of type `Hand`. The message is a `String`. These values are stored in instance variables in the canvas class. 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 canvas 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 canvas 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()
    // if 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;
    repaint();
}

```

The `doHigher()` and `doLower()` methods are almost identical (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 applet 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 `paint()` method of the applet 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 `paint()` 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](#). (This file contains the source for both the applet class, `HighLowGUI` and the canvas class `HighLowCanvas`.)

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 Choice menu at the bottom 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 Choice menu, and then the drawing area is cleared. This lets you draw in cyan on a magenta background if you have a mind to.

Sorry, but your browser
doesn't support Java.

The drawing area in this applet is a component, belonging to the class `SimplePaintCanvas`. I wrote this class as a sub-class of `Canvas` and programmed it to listen for mouse events and respond by drawing a curve. As in the `HighLowGUI` applet, all the action takes place in the canvas class. The main applet class just does the set up. One new feature of interest is the Choice menu. This component is an object belonging to the standard class, `Choice`. 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 Choice menu.**

You'll find the source code for this example in the file [SimplePaint2.java](#). The file contains both classes, the applet class `SimplePaint2` and the canvas class `SimplePaintCanvas`.

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

Section 6.7

Looking Back: The Java 1.0 Style of Event Handling

WHEN JAVA 1.1 WAS RELEASED, it included a large number of changes from Java 1.0. One of the biggest changes was the introduction of an entirely new system for handling events. It is the newer Java 1.1 style of event handling that I have been discussing in this chapter. The newer style is also used in the most recent versions of Java, and it will almost certainly continue to be used in the future. The new style of event handling is more flexible and more efficient than the old style. Unfortunately, it is also more complicated. The big advantage of Java 1.0 event handling is that it isn't necessary to set up listeners for events. Its big disadvantage is that you are not able to set up listeners -- but the disadvantage only becomes apparent in projects that use more than just a few classes and objects. For small projects, it can still make sense to use Java 1.0 style event handling, if you want to go through the trouble of learning a whole new event-handling architecture. It is also true that a few people out there are still using older browsers that support Java 1.0 but not Java 1.1. If you want those people to be able to use your Java programs, you need to write them in Java 1.0.

The features of Java 1.0 that are superceded by newer features of Java 1.1 are said to be **deprecated**. Deprecated features are still part of the language, but their use is discouraged. The idea is that they might be dropped from Java altogether in some future version (although I don't think that is likely in the near term.) The whole Java 1.0 event-handling architecture is deprecated in Java 1.1 and beyond.

In this section, I'll give an outline of the Java 1.0 style of event handling. For more complete information, see a Java reference or look at Chapters 5 and 6 in the first edition of this on-line text, which should still be available on the Web at <http://math.hws.edu/eck/cs124/javanotes1/index.html>.

One important note: You can use either Java 1.0 or Java 1.1 style event handling. But **you can't mix them in the same applet or program**. For a given project, you have to decide which model you want to use and stick to it.

The Event Model in Java 1.0

In Java 1.0 there is just one class of events, `java.awt.Event`. Every event is generated by a component, which is called the **target** of the event. The target of an event object, `evt`, is given by the `public` instance variable `evt.target`. Other instance variables carry other information about various types of events. For example, the coordinates of a mouse-related event are given by `evt.x` and `evt.y`.

In Java 1.1 style event handling, an event is sent only to objects that are listening for the event. (This is where the Java 1.1 style gets its advantage in efficiency: Events are not processed unless they have been **enabled**, usually by registering a listener.) In the Java 1.0 style, whenever any event occurs, the system calls a method named `handleEvent()` in the target component. For many types of events, `handleEvent()` will in turn call a special purpose event-handling routine such as `mouseDown()` or `keyDown()`. Some of these event-handling routines are discussed below. The `handleEvent()` method might or might not actually handle the event. It returns a boolean value to the system to indicate whether the event was handled. If the event is not handled by the target component, and if that component is contained in some other component -- such as a `Panel` or `Applet` -- then the system gives the container component a chance to handle the event by calling the `handleEvent()` method of the container. This process can continue up a chain of containment until a top-level component such as an applet or frame is reached. If the top-level component doesn't handle the event, then the event is ignored.

This might sound very complicated but what it comes down to in most cases is this: Mouse and keyboard events are handled by the component to which they are targeted (usually a canvas object, if not the applet

itself). The top-level applet class does all the other event-handling for the program. Events from buttons, text fields, choice menus, and so on are allowed to filter up to the top level where they are handled by a method in the applet class. (In fact, in any case where this simplified model is not appropriate, you should really be using Java 1.1 style event handling.)

To deal with certain types of events, such as those generated by scroll bars, it is necessary to override the `handleEvent()` method itself. But for the more common types of events, there are special purpose event-handling methods that you can override.

Mouse and Keyboard Events in Java 1.0

As the user moves the mouse and presses buttons on the mouse, several event-handling methods can be called. The most useful mouse-related methods are:

```
public boolean mouseDown(Event evt, int x, int y) {
    . . . // respond to fact that user has pressed mouse button
    return true;
}
public boolean mouseUp(Event evt, int x, int y) {
    . . . // respond to fact that the user has released mouse button
    return true;
}
public boolean mouseDrag(Event evt, int x, int y) {
    . . . // respond to fact that mouse has moved, while
           // user is holding down a mouse button
    return true;
}
public boolean mouseMove(Event evt, int x, int y) {
    . . . // respond to fact that mouse has moved, while
           // user is NOT holding down a mouse button
    return true;
}
```

Note that these are boolean-valued methods. In almost all cases, you should return `true`, which indicates that you have handled the event. In the rare cases where you want to let the event be passed on to the next possible handler, return `false` instead.

If the user clicks the mouse somewhere in the rectangle occupied by a component, the `mouseDown()` method is called when the user presses the button, and the `mouseUp()` method when the user releases it. If the user moves the mouse while holding the button down, the computer will call `mouseDrag()` over and over as the mouse moves. The `mouseMove()` method is called when the user moves the mouse without holding a button down.

In all of these methods, the parameters `x` and `y` give the horizontal and vertical position of the mouse, in coordinates appropriate to the component. The parameter `evt` carries full information about the event that caused the method to be called. (The `x` and `y` coordinates have been pulled out of this `Event` object for your convenience.) You can check whether the user was holding down the shift key, the control key, or the Meta key when the event occurred by calling the boolean-valued methods `evt.shiftDown()`, `evt.controlDown()`, and `evt.metaDown()`. Holding down the Meta key is equivalent to pressing the right mouse button, so `evt.metaDown()` also returns `true` if the right mouse button is down. (For some reason, there is no method for testing whether the Alt key -- or middle mouse button -- is down. Instead you have to test "if ((`evt.modifiers` & `Event.ALT_MASK`) != 0)!"

The applet at the bottom of this page uses Java 1.0 style handling of mouse events. If you are interested, you can find the source code for this applet in the file [TrackLines.java](#).

As for keyboard events in Java 1.0, every time the user presses a key on the keyboard, two events are generated: one when the user presses the key and one when the user releases the key. A component can be programmed to respond to these events by overriding the `keyDown()` and `keyUp()` methods, which are defined as follows:

```
public boolean keyDown(Event evt, int key) {
    . . . // respond to the fact that a key has been pressed
    return true;
}
public boolean keyUp(Event evt, int key) {
    . . . // respond to the fact that a key has been released
    return true;
}
```

The `key` parameter in these methods tells you which key was pressed by the user. You might be surprised to see that this parameter has type `int` rather than `char`. This is because characters aren't the only things that the user can type! The user can also press "action keys" such as the arrow keys and the function keys F1, F2, etc.

If the user actually types a character, then the `key` parameter tells you which character was typed. Because `key` is of type `int`, you have to use a type-cast to discover which character was typed:

```
char typedChar = (char)key;
```

If the user pressed one of the action keys, then the value of the `key` parameter will be a special constant value that specifies which key was pressed. The value will be one of the following predefined constants: `Event.UP`, `Event.DOWN`, `Event.LEFT`, `Event.RIGHT`, `Event.HOME`, `Event.END`, `Event.PGUP`, `Event.PGDN`, or `Event.F1` through `Event.F12`. The first four of these, which are probably the most useful, correspond to the up, down, left, and right arrow keys.

When dealing with keyboard events, there is the complication of having to worry about the input focus. A component that processes keyboard events should keep track of whether it has the input focus, so that it can change its appearance when it has the focus. A focus event is sent to a component whenever it gains or loses the input focus. In Java 1.0, this means calling the event-handling methods "`public boolean gotFocus(Event evt, Object what)`" and "`public boolean lostFocus(Event evt, Object what)`". You can override these methods in a component that needs to respond to focus events. In general, both of the parameters to these methods can be ignored.

Action Events in Java 1.0

Besides mouse and keyboard events, there are the events that are generated when the user interacts with graphical interface components such as buttons, menus, and text boxes. (The user actually causes these other events with the mouse or keyboard, but they are translated by the system into more meaningful terms.) For many of these events, the system calls the `action()` method, which takes the form

```
public boolean action(Event evt, Object arg) {
    . . . // respond to the action event
    return true; // or return false, if action not handled
}
```

To deal with action events, you should override the `action()` method in your subclass of `Applet`. The method should handle all the action events from any component contained in the applet. When you write an `action()` method, you know exactly which components might generate action events for it to handle, so you can write it to handle just those components.

The second parameter in the `action()` method, `arg`, contains some information relevant to the event. What type of information `arg` contains depends on the type of object that generated the event. The first parameter, `evt`, has an instance variable, `evt.target`, which tells which component first received the event. For an action event, this is also the component that generated the event in the first place. This means that an `action()` method often looks something like this:

```
public boolean action(Event evt, Object arg) {
    if (evt.target == button1) {
        // handle a click on button1
        return true;
    }
    else if (evt.target == button2) {
        // handle a click on button2
        return true;
    }
    else if (evt.target == colorChoice) {
        // handle a selection from Choice object colorChoice
        return true;
    }
    .
    .    // handle other possible targets
    .
    else
        return super.action(evt, arg);
}
```

(In simple cases, when there are only a few components to worry about, you might not even have to check `evt.target`; the second parameter, `arg`, might contain all the information you need.)

Here are the details about the action events that can be generated by various types of components:

- A **Button** generates an action event when the user clicks on the button. The `arg` parameter of `action()` is a string that is equal to the button's label. (If you want to use this parameter, you can type cast `arg` to type `String`.)
- A **Choice** generates an action event when the user selects one of the items in the Choice menu. The `arg` parameter is the text of the item that the user selected.
- A **TextField** generates an action event if the user presses return while typing in the input box. The `arg` parameter is a string that gives the contents of the box.
- A **Checkbox** generates an action event when the user clicks on it. The `arg` is an object of type `Boolean` that gives the new state of the box. (An object of type `Boolean` contains a value belonging to the primitive type `boolean`. A `boolean` value is not an object. "Wrapping" it inside a `Boolean` object makes it possible to treat it as an object. To get the `boolean` value contained in `arg`, you could say `boolean newState = ((Boolean)arg).booleanValue();`. It's generally much easier to use the `Checkbox`'s `getState()` method to find out its state.)

Although this brief overview of events in Java 1.0 does not give you enough information to write complex applications using the Java 1.0 event model, that's not something you should be doing anyway. There is probably enough information on this page for you to use Java 1.0 event handling in simple applets.

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 "canvas", and place a "Roll" button below the canvas. 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, rather than putting it in a "buttonBar" panel and adding the panel to the applet.)

[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 `SimpleAnimationApplet`, 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 are 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*?

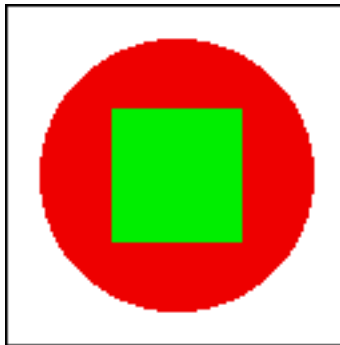
Question 3: Explain carefully what the `repaint()` method does.

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; i <= 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 `Canvas`. What is the point of this class? How are canvas objects used, and why?

Chapter 7

Advanced GUI Programming

THE JAVA PACKAGES `java.awt` and `java.awt.event` contain classes for writing programs that use a graphical user interface. The previous chapter introduced several of these classes, such as the class `Button`. An object of type `Button` represents a push-button that the user can click to perform some action. When the programmer creates an instance of this class, it will appear on the screen as a button appropriate to the platform on which the program is running. Even though the button will appear different on different platforms, its "logical" or "abstract" behavior will be the same. The Java programmer only has to worry about this abstract behavior; the platform-dependent details are left to the Java implementation on each platform. This is why the Java GUI system is called the **Abstract Windowing Toolkit (AWT)**.

In this chapter, we'll take a more detailed look at using the AWT for graphical user interface programming, starting with some advanced 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. Two new features of Java, threads and nested classes, will be introduced. Both of these features are useful in many applications besides GUI programming.

This textbook is based on Java 1.1. A newer version, Java 1.2, builds on Java 1.1 by adding a large number of new standard classes. In particular, Java 1.1 introduced a new set of user interface components called **Swing**, as a supplement to the AWT. Java 1.2, together with a few optional features, is sometimes referred to as Java 2 or Java Platform 2. The last section of this Chapter is a brief survey of Swing and Java 2.

The material in this chapter will be used in a number of examples and programming exercises in future chapters. Aside from that, the material in this chapter is not a prerequisite the rest of this textbook.

Contents Chapter 7:

- Section 1: [More about Graphics](#)
- Section 2: [More about Layouts and Components](#)
- Section 3: [Standard Components and Their Events](#)
- Section 4: [Programming with Components](#)
- Section 5: [Threads, Synchronization, and Animation](#)
- Section 6: [Nested Classes and Adapter Classes](#)
- Section 7: [Frames and Dialogs](#)
- Section 8: [Looking Forward: Swing and Java 2](#)
- [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 representing 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 (and possibly in other types of image files, depending on the version of Java). For example, suppose that the image of an ace of clubs, shown at the right, is contained in a file named "ace.gif". In the source code for an applet, if `img` is a variable of type `Image`, you could say

```
img = getImage( getCodeBase(), "ace.gif" );
```



to 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. Suppose that `g` is a graphics context, that is, an object belonging to the class `Graphics`, and suppose that `img` is a variable of type `Image`. Then the usual command for drawing the image, `img`, in the graphics context, `g`, is

```
g.drawImage(img, x, y, this);
```

This command can be used in an instance method of an applet, canvas, or other component. The parameters `x` and `y` are integers that give the position of the top-left corner of the displayed image. The fourth parameter, `"this"`, requires some explanation. It's there because of the funny way that Java works with images from image files. 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 `Component` object can act as an `ImageObserver`, including applets and canvases. In `"g.drawImage(img, x, y, this);"`, the special variable `this` refers to the object whose source code you are writing. When you are drawing an image to the screen, you should almost always use `"this"` as the fourth parameter to `drawImage()`.

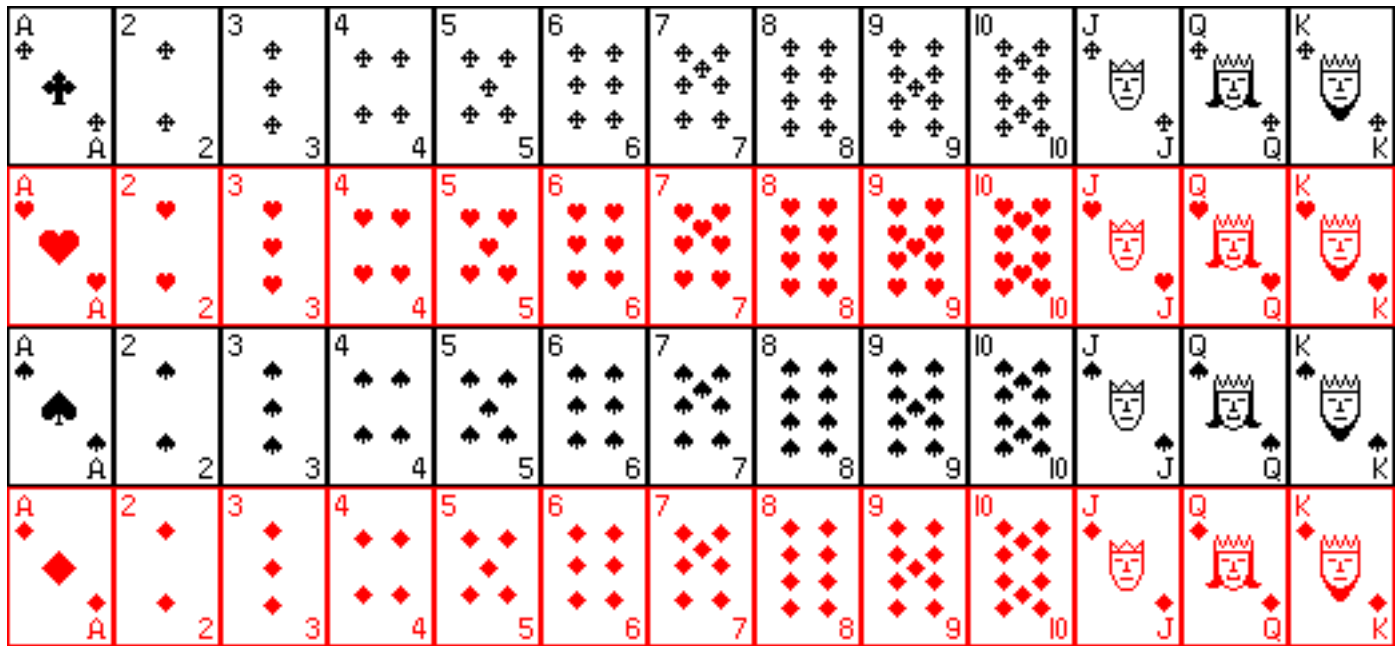
There are a few useful variations of the `drawImage()` command. For example, it is possible to scale the image as it is drawn to a specified width and height. This is done with the command

```
g.drawImage(img, x, y, width, height, this);
```

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 could 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](#):

Sorry, but your browser
doesn't support Java.

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);
        System.out.println(card.toString());
    }
} // end drawCard()

```

The complete source code for this applet can be found in [HighLowGUI2.java](#).

Double Buffering for Smooth Animation

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. If this technique is used for repainting the screen, then behind the scenes, in memory, an old image is erased and a new one is drawn step-by-step. This takes some time. If all this drawing were done on screen, the user would see the image flicker. 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 can be used to do smooth, flicker-free animation and dragging.

I call an image in memory an **off-screen canvas**. The technique of drawing to an off-screen canvas and then quickly copying the canvas 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 as an off-screen canvas to prepare 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 entirely flicker-free.)

Here are two applets that are identical, except that one uses double buffering and one does not. You can drag the red and blue squares around the applets. For the applet on the left, you should notice an annoying flicker as you drag a square (although on very fast computers it might not be all that noticeable):

Sorry, but your browser
doesn't support Java.

An off-screen `Image` can be created by calling the instance method `createImage()`, which is defined in the `Component` class. You can use this method in applets and canvases, for example. The `createImage()` method takes two parameters to specify the width and height of the image to be created. For example,

```
Image OSC = createImage(width, height);
```

Drawing to an off-screen canvas 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 canvas. (This works only for off-screen canvases. If you try to do this with an `Image` from a file, an error will occur.) That is, if `OSC` is a variable of type `Image` that refers to an off-screen canvas, you can say

```
Graphics offscreenGraphics = OSC.getGraphics();
```

Then, any drawing operations performed with the graphics context `offscreenGraphics` are applied to the off-screen canvas. For example, `"offscreenGraphics.drawRect(10,10,50,100);"` will draw a 50-by-100-pixel rectangle on the off-screen canvas. Once a picture has been drawn on the off-screen canvas, the picture can be copied into another graphics context, `g`, using the method


```
g.drawImage(OSC).
```

When using an off-screen canvas to avoid flicker, it's convenient to manage the off-screen canvas in the `update()` method, which is called by the system when a component needs to be repainted. The `update()` method can create the off-screen canvas if it doesn't already exist, call the component's `paint()` method to draw to the off-screen canvas, and then copy the contents of the off-screen canvas to the screen. With just a little more work, we can even allow for the case where the size of the component can change. Here is what you would put in your applet or canvas class to make this work:

```
/* Some variable used for double-buffering */

Image OSC; // The off-screen canvas (created and used in update()).
           // The size of the OSC matches the size of the component.

int widthOfOSC, heightOfOSC; // Current width and height of OSC.
                             // 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 OSC is created.

public void update(Graphics g) {

    // To implement double-buffering, the update method calls paint to
    // draw the contents of the applet on an off-screen canvas. Then
    // the canvas is copied onto the screen. This method is responsible
    // for creating the off-screen canvas. It will make a new OSC if
    // the size of the applet changes.

    if (OSC == null || widthOfOSC != getSize().width
        || heightOfOSC != getSize().height) {
        // Create the OSC.
        // (Or make a new one if applet size has changed.)
        OSC = null; // (If OSC already exists, this frees up the memory.)
        OSC = createImage(getSize().width, getSize().height);
        widthOfOSC = getSize().width;
        heightOfOSC = getSize().height;
    }

    /* Set things up in the OSC the way things are usually set
       up for the paint method: Clear the OSC to the background color.
       Set the graphics context to use the component's drawing color and
       font.
    */

    Graphics OSGr = OSC.getGraphics(); // Graphics context
                                     // for drawing to OSC.
    OSGr.setColor(getBackground());
    OSGr.fillRect(0, 0, widthOfOSC, heightOfOSC);
    OSGr.setColor(getForeground());
    OSGr.setFont(getFont());

    paint(OSGr); // Draw component's contents to OSGr
                // instead of directly to g.
    OSGr.dispose(); // We're done with this graphics context.
```

```

        g.drawImage(OSC,0,0,this);    // Copy OSC to screen.

    } // end update()

    public void paint(Graphics g) {
        // Draw the contents of the applet to the graphics context g,
        // just as they would be drawn on the screen.
        .
        .
    } // end paint()

```

This is the technique used in the applet, shown above, that uses double buffering for smooth dragging. You can find the complete source code in the file [DoubleBufferedDrag.java](#). An update method and a few extra instance variables are the only difference between the double-buffered version and the non-double-buffered version, [NonDoubleBufferedDrag.java](#). The same technique can be used to do smooth animation, as we'll see in [Section 5](#).

Double Buffering for Screen Repainting

Flicker was only one of the problems that we had with drawing in the previous chapter. Another problem was that, in many cases, we had no convenient way of remembering the contents of a component so that we could redraw the component 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. Double buffering can be used to solve this problem too. The idea is simple: Keep a copy of the drawing in an off-screen canvas. When the component needs to be redrawn, copy the off-screen canvas 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 canvas should always contain a copy of the picture on the screen. The `update()` and `paint()` methods should do nothing but copy the off-screen canvas 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 canvas. 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 canvas, 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.)

There are two approaches to keeping the image on the screen synchronized with the image in the off-screen canvas. In the first approach, in order to change the image, you make that change to the off-screen canvas 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 canvas 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.

In this application, it doesn't make sense to have the `update()` method create the canvas since the off-screen canvas is used outside the `update()` method. I suggest having a separate method to handle the creation of the off-screen canvas and its re-creation when the size of the component changes. This method should always be called before using the off-screen canvas in any way. Here is the basic code that a class needs in order to implement this:

```

/* Some variables used for double-buffering. */

```

```

Image OSC; // The off-screen canvas (created in setupOSC()).

int widthOfOSC, heightOfOSC; // Current width and height of OSC.
                                // 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 OSC is created.
                                // The picture in the off-screen canvas
                                // is lost when that happens.

void setupOSC() {
    // This method is responsible for creating the off-screen canvas.
    // It should always be called before using the OSC. It will make a
    // new OSC if the size of the applet changes. A new off-screen
    // canvas is filled with the background color of the component.
    if (OSC == null || widthOfOSC != getSize().width
        || heightOfOSC != getSize().height) {
        // Create the OSC, or make a new one
        // if component size has changed.
        OSC = null; // (If OSC already exists, this frees up the memory.)
        OSC = createImage(getSize().width, getSize().height);
        widthOfOSC = getSize().width;
        heightOfOSC = getSize().height;
        Graphics OSGr = OSC.getGraphics();
        OSGr.setColor(getBackground());
        OSGr.fillRect(0, 0, widthOfOSC, heightOfOSC);
        OSGr.dispose()
    }
}

public void update(Graphics g) {
    // Redefine update so it doesn't clear before calling paint().
    paint(g);
}

public void paint(Graphics g) {
    // Just copy the off-screen canvas to the screen.
    setupOSC(); // Ensure that OSC exists first!!
    g.drawImage(OSC, 0, 0, this);
}

```

Note that the contents of the off-screen canvas are lost if the size changes. If this is a problem, you can consider copying the contents of the old off-screen canvas to the new one before discarding the old canvas. 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.

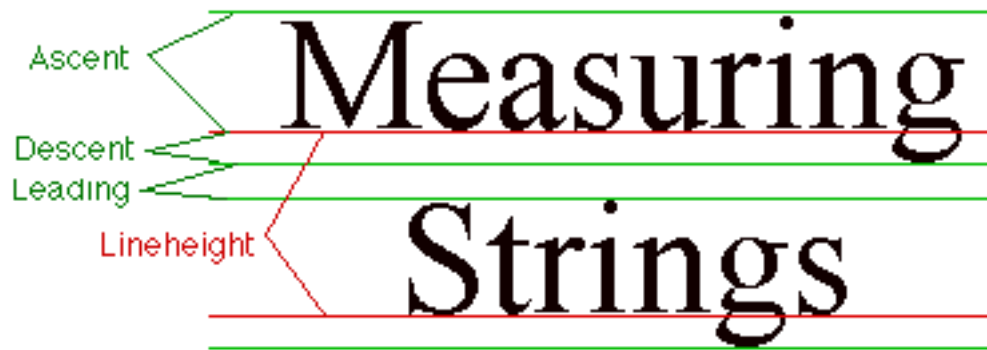
FontMetrics

In the rest of this section, we turn from Images to look briefly at a few other aspects 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 `paint()` method that shows the message "Hello World" in the exact center of the component:

```
public void paint(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);
}
```

Drawing with XORMode

Ordinarily, when shapes or text are drawn in a graphics context, `g`, the colors of the affected pixels are changed to the current drawing color of `g`, as specified by `g.setColor()`. In Java, this type of drawing is called **paint mode**, and it is not the only possibility. There is another mode of drawing called **XOR mode**, in which the effect on the color of pixels is not so straightforward. Drawing in XOR mode has an interesting and useful property: If you perform exactly the same drawing operation twice in a row, the second operation reverses the effect of the first, leaving the image in its original state. Unfortunately, you can't be sure what colors will be used when you draw in XOR mode.

XOR mode can be used, for example, to implement a "rubber band cursor." A rubber band cursor is commonly used to draw straight lines. When the user clicks and drags the mouse, a moving line stretches between the starting point of the drag and the current mouse location. When the user releases the mouse, the line becomes a permanent part of the image. While the mouse is being dragged, the line is drawn in XOR mode. When the mouse moves, the line is first redrawn in its previous position. In XOR mode, this second drawing operation erases the first. Then, the line is drawn in its new position. When the user releases the mouse, the line is erased one more time and is then drawn permanently using paint mode. The same idea can be used for other figures besides lines. However, it doesn't work very well for filled shapes because of the weird color effects. (Of course, maybe you like weird color effects.) A simple solution to the problem with filled shapes is to draw only the outline of the shape when dragging.

Here is a little applet that illustrates XOR mode. Draw straight red lines by clicking and dragging. Draw blue filled rectangles by right-clicking and dragging (or, on the Mac, Command-clicking). Shift-click on the applet to clear it. Check out the colors when you draw one rectangle on top of another:

Sorry, but your browser
doesn't support Java.

If `g` is a graphics context, then you can use the command `g.setXORMode(xorColor)` to start using XOR mode. The `xorColor` parameter is a `Color` that will (on some platforms) be used as follows: If you draw in XOR mode over pixels whose color is the specified `xorColor`, then those pixels will be changed to the current drawing color of the graphics context. That is, drawing over the `xorColor` in XOR mode is the same as drawing over this color in the regular paint mode. In almost all cases, you want to use the background color as the `xorColor`, so you should say `g.setXORMode(getBackground())`. You can switch from XOR mode back to regular paint mode with the command `g.setPaintMode()`.

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, top, width, and 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.
}

```

The source code for the above applet is in the file [RubberBand.java](#).

A Better Paint Program

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 canvas to save a copy of the user's work. 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. The shapes are drawn using rubber band cursors in XOR mode. Try it out! Check that when you cover up the applet with another window, your drawing is still there when you uncover it.

Sorry, but your browser
doesn't support Java.

The source code for this improved paint applet is in the file [SimplePaint2.java](#). It uses an off-screen canvas pretty much in the way described above. The `paint()` method for the `Canvas` object simply copies the off-screen canvas to the screen. When the user begins a drag operation, two graphics contexts are obtained, one for drawing on the screen and one for drawing to the off-screen canvas. If the user is sketching a curve, every line segment in the curve is drawn to both the screen and to the off-screen canvas. This keeps the off-screen picture in sync with the picture on the screen. When the user is drawing a shape, the rubber band cursor is treated somewhat differently. The rubber band cursor is not a permanent part of the image, so there is no need to draw it to the off-screen canvas. It is drawn only on the screen. When the drag operation ends, the final shape is drawn to both the off-screen canvas and to the screen. There are lots of other details to attend to. I encourage you to read the source code and see how its done.

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

Section 7.2

More about Layouts and Components

MANY OF THE CLASSES IN THE AWT represent visual elements of a graphical user interface, such as buttons, text-input boxes, and menus. All these classes, except for a few related to menus, are subclasses of the class `java.awt.Component`.

`Component` is an abstract class, so that you can only create objects belonging to its subclasses, not to `Component` itself. The subclasses that represent standard GUI elements are: `Button`, `Checkbox`, `Choice`, `Label`, `List`, `Scrollbar`, `TextArea`, and `TextField`. The `Canvas` class, which we've used in several examples, is also a subclass of `Component`. Objects of these classes have predefined behaviors. For the most part, all you have to do is add them to your program and they take care of themselves. When the user interacts with one of these components and some action is required, the component generates an event that your program can detect and react to. I will discuss the standard GUI components in the [next section](#).

To appear on the screen, a component must be added to a container. A container in this context is a component that can contain other components. Containers are represented by the class `java.awt.Container`, which is a subclass of `Component`. The `Container` class, like `Component`, is an abstract class. `Container` has two direct subclasses, `Window` and `Panel`. A `Window` represents an independent top-level window that is not contained in any other component. `Window` is not really meant to be used directly. It has two subclasses: `Frame`, to represent ordinary windows that can have their own menu bars, and `Dialog` to represent dialog boxes that are used for limited interactions with the user. I will discuss the window classes in [Section 7](#).

A `Panel`, on the other hand, is a container that does not exist independently. The `Applet` class is a subclass of `Panel`, and as you have seen, an applet does not exist on its own; it must be displayed on a Web page or in some other window. Any panel must be contained inside something else, either a `Window`, another `Panel`, or -- in the case of an `Applet` -- a page in a Web browser.

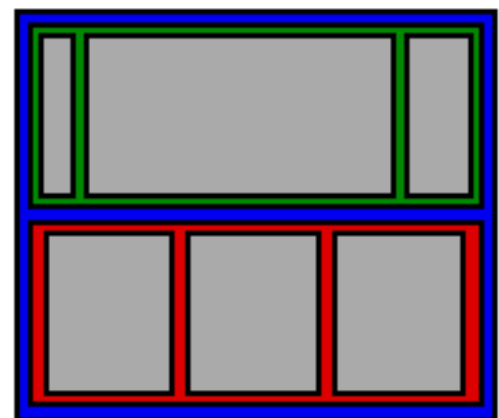
The fact that panels can contain other panels means that you can have many levels of components containing other components, as shown in the illustration on the right. This leads to two questions: How are components added to a container? How are their sizes and positions controlled?

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 in the AWT:

`FlowLayout`, `GridLayout`, `BorderLayout`, `CardLayout` and `GridBagLayout`. 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 `Panels`, including `Applets`, the default layout manager belongs to the class `FlowLayout`. For `Windows`, the default layout manager is a `BorderLayout`.

You can change the layout manager of a container using its `setLayout()` method. It is 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 option further in [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`



Three Panels, shown in color, containing six additional Components, shown in gray.

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 each of the 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 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 want an applet to contain one button, located in the upper right corner of the applet. The default layout manager for an applet is a `FlowLayout` that uses center alignment. It would center the button horizontally. We need to give the applet a new layout manager that uses right alignment, which will shove the button to the right edge of the applet. The following `init()` method will do this:

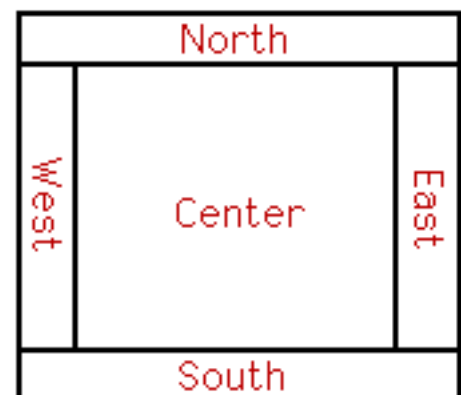
```
public void init() {
    setLayout( new FlowLayout(FlowLayout.RIGHT, 5, 5) );
    add( new Button("Press me!") );
}
```

Although `FlowLayouts` are useful in certain circumstances, I almost always set the layout manager of an applet to be a `BorderLayout` or a `GridLayout`. I use the `FlowLayout` mainly when I want to use a panel to hold a strip of controls along the top or bottom of an applet. The examples in [Section 6.6](#) used panels in this way.

BorderLayout

A `BorderLayout` places one component in the center of a container. The 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`. For example, the following code creates a panel with `drawArea` as its center component and with scroll bars to the right and below:

```
Panel panel = new Panel();
panel.setLayout(new BorderLayout());
    // To use BorderLayout with a Panel, you have
    //      to change the panel's layout manager; otherwise,
    //      a FlowLayout is used.
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.)

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.

#1	#2
#3	#4
#5	#6

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:

```
Panel buttonBar = new Panel();
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 `Panel` that can show any one of three `Panels`: `panel1`, `panel2`, and `panel3`. Assume that `panel1`, `panel2`, and `panel3` have already been created:

```
cardPanel = new Panel();
// 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.

Components in Applets

When you are writing an Applet, you should remember that applets are themselves containers. This means that they have `add()` methods that can be used to add components and a `setLayout()` method that can be used to replace the default layout manager. In general, the entire layout of an applet should be set up in its `init()` method. This will often involve constructing sub-panels and adding them to the applet.

One final point: With most layout managers, you can specify horizontal and vertical gaps between components. But what if you want gaps between the edges of a container and the components that it contains? For that, you have to override the `getInsets()` method of the container. For an applet or panel, the definition of this method usually has the form:

```
public Insets getInsets() {
    return new Insets(top, left, bottom, right);
}
```

where `top`, `left`, `bottom`, and `right` are integers that specify the number of pixels to be inserted as a gap along each edge. However, things get a little more complicated in the case of a container that already has non-zero insets -- most commonly a window that uses insets to leave space for the borders of the window and the menu bar.

In [Section 7](#), I'll explain how to use components and insets in independent windows that belong to the classes `Frame` and `Dialog`.

An Example

To finish this section, here is an applet that demonstrates various layout managers:

Sorry, but your browser
doesn't support Java.

The applet itself uses a `BorderLayout` with vertical gaps of 3 pixels. These gaps show up in blue, which is the background color of the applet as a whole. The blue border around the edges comes from a `getInsets()` method in the applet. The Center component of the applet is a panel. This panel is set to use a `CardLayout` as its layout manager. The layout contains six 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 `Choice` menu, which contains the names of the six 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 `Label` that displays an appropriate message whenever the user clicks on a button or chooses an item from the `Choice` menu.

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.

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

Section 7.3

Standard Components and Their Events

THIS SECTION DISCUSSES some of the GUI interface elements that are represented by subclasses of `Component`. It also introduces the event classes and listener interfaces associated with each type of component. The treatment here is very brief. I will give some examples of programming with these components in the [next section](#).

The `Component` 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 component. 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. 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.
- `comp.getParent()` is a function that returns a value of type `Container`. The container is the one that contains the component, if any. For a top-level component such as a `Window` or `Applet`, 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`.
- `comp.setEnabled(true)` and `comp.setEnabled(false)` can be used to enable and disable the component. This is only useful for certain types of components, such as `button`. When a button is disabled, its appearance changes, and clicking on it will have no effect. 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, it inherits the colors of its parent container. The command `comp.setFont(font)` sets the default font that is used for text displayed on the component. (These should work for all components, but might not work properly for some of the standard components, depending on the version of Java.)

There is also an event type associated with components. Whenever a component is hidden, shown, moved, or resized, an event belonging to the class `ComponentEvent` is generated. These events are handled in the same way as the mouse and keyboard events that we saw in the previous chapter: If you want to listen for a `ComponentEvent`, you have to implement the `ComponentListener` interface. This interface defines four methods:

```
public void componentResized(ComponentEvent evt);
public void componentMoved(ComponentEvent evt);
public void componentHidden(ComponentEvent evt);
public void componentShown(ComponentEvent evt);
```

Once you have an object that implements this interface, you must register it with a component by calling the component's `addComponentListener()` method. The parameter, `evt`, contains a function, `evt.getComponent()`, that returns the `Component` that generated the event. From this interface, you are most likely to use the `componentResized()` method, which is useful when you want to take some

action each time the size of a component is changed.

For the rest of this section, we'll look at subclasses of `Component` 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.

Here is a simple applet that demonstrates several of the components discussed in this section. Across the top are a `Choice` menu, a `Checkbox`, and a `TextField`. The rest of the applet shows a large canvas with `ScrollBars` below and to the right. The scroll bars control the color of the display. If you type in the `TextField` and press return, the text you typed will be displayed on the canvas.

Sorry, but your browser
doesn't support Java.

Although this is a rather silly example, it does show how to create and use several types of components. You'll find the source code for this example in the file [EventDemo.java](#).

The Button Class

An object of class `Button` is a push button. You've already seen buttons used in the previous chapter, but we can use a review of `Buttons` as a reminder of what's involved in using components, events, and listeners. (Some of the methods described here are new.)

- **Constructors:** The `Button` class has a constructor that takes a string as a parameter. This string becomes the label displayed on the button. For example: `stopGoButton = new Button("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 label that is displayed on the button.
- **Component methods:** There are several useful methods in the `Button` class. For example, `stopGoButton.setLabel("Stop")` changes the label displayed on the button to "Stop". And `stopGoButton.setActionCommand("sgb")` changes the action command associated to this button for action events.

Of course, `Buttons` also have all the general `Component` methods, such as `setEnabled()` and `setFont()`. The `setEnabled()` and `setLabel()` 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!"

The Label Class

`Labels` are certainly the simplest type of component. An object of type `Label` 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 `Label` specifies the text to be displayed:

```
Label message = new Label("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 `Label.LEFT`, `Label.CENTER`, and `Label.RIGHT`. For example,

```
Label message = new Label("Hello World!", Label.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 `Label` class is a subclass of `Component`, you can use methods such as `setForeground()` with labels. For example:

```
Label message = new Label("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
```

The Checkbox and CheckboxGroup Classes

A `Checkbox` is a component that has two states: checked and unchecked. 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 checked and `false` if the box is unchecked. A checkbox has a label, which is specified when the box is constructed

```
Checkbox showTime = new Checkbox("Show Current Time");
```

Usually, it's the user who sets the state of a `Checkbox`, but you can also set the state in your program. The current state of a checkbox is set using its `setState(boolean)` method. For example, if you want the checkbox `showTime` to be checked, you would say `showTime.setState(true);`. To uncheck the box, say `showTime.setState(false);`. You can determine the current state of a checkbox by calling its `getState()` 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 `getState()` 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, it generates an event of type `ItemEvent`. The associated listener interface is `ItemListener`. To receive notification of item events from a checkbox, an object must implement the `ItemListener` interface and it must be registered with the checkbox by calling its `addItemListener()` method. The `ItemListener` interface defines one method, `public void itemStateChanged(ItemEvent evt)`.

For handling `ItemEvents`, I recommend calling `evt.getSource()` in the `itemStateChanged()` 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 `getSource()` method can be used with any event type, not just `ItemEvents`. The method `evt.getSource()` returns the object that generated the event. 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 `itemStateChanged()` method might look like this:

```
public void itemStateChanged(ItemEvent evt) {
    Object source = evt.getSource();
    if (source == cb1) {
        boolean newState = cb1.getState();
        ... // respond to the change of state
    }
    else if (source == cb2) {
        boolean newState = cb2.getState();
        ... // respond to the change of state
    }
}
```

Closely related to checkboxes are **radio buttons**. Radio buttons occur in groups. At most one radio button in a group can be checked at any given time. In Java, a radio button is just an object of type `Checkbox` that is a member of such a group. An entire group of radio buttons is represented by an object belonging to the class `CheckboxGroup`. The class `CheckboxGroup` has methods

```
public void setSelectedCheckbox(Checkbox box);
public Checkbox getSelectedCheckbox();
```

for selecting one of the checkboxes in the group, and for discovering which box is currently selected. The `getSelectedCheckbox()` method will return `null` if none of the boxes in the group is currently selected.

To create a group of radio buttons, you should first create an object of type `CheckboxGroup`. To create the individual buttons, use the constructor

```
Checkbox(String label, CheckboxGroup group, boolean state);
```

The third parameter of this constructor specifies the initial state of the checkbox. Remember that at most one of the checkboxes in the group can have its state set to `true`. For example:

```
CheckboxGroup colorGroup = new CheckboxGroup();
Checkbox red    = new Checkbox("Red", colorGroup, false);
Checkbox blue   = new Checkbox("Blue", colorGroup, false);
Checkbox green  = new Checkbox("Green", colorGroup, true);
Checkbox black  = new Checkbox("Black", colorGroup, false);
```

This creates a group of four radio buttons labeled "Red", "Blue", "Green", and "Black". Initially, the third button is selected. You still have to add the `Checkboxes`, individually, to some container. The `CheckboxGroup` itself is not a component and cannot be added to a container. To actually use the buttons effectively in your program, you would presumably want to store either the `CheckboxGroup` object or the individual `Checkbox` objects in instance variables. `ItemEvents` are generated by each individual `Checkbox` object when its state changes. Again, you can often ignore these events and simply ask the `CheckboxGroup` which `Checkbox` is selected, whenever you have a need to know. If you do want to respond to `ItemEvents`, you need to register your listener with each of the `Checkboxes` individually.

The Choice Class

A `CheckBoxGroup` is one way of allowing the user to select one option from a predetermined set of options. The `Choice` class represents another way of doing the same thing. An object of type `Choice`, however, presents the options in the form of a menu. The menu lists a number of items. Only the selected item is displayed. However, when the user clicks on the `Choice` component, the entire list is displayed, and the user can select one of the items from the list. (There are three types of menus in Java: `Choice` menus, pop-up menus, and pull-down menus. Pop-up menus and pull-down menus are not `Components`. I'll discuss pop-up menus later in this section and pull-down menus in [Section 7](#). In fact, `Choice` components are not technically considered to be menus, but it's hard to find another word that adequately describes what they do.)

When a `Choice` object is first constructed, it initially contains no items. An item is added to the bottom of the menu by calling its instance method, `add(String)`. The `getItemCount()` method returns an `int` that gives the number of items in the list, and the method

```
public String getItem(int index)
```

gets the item in position number `index` in the list. (Items are numbered starting with zero, not one.) Other useful methods are:

```
public int getSelectedIndex();
public String getSelectedItem();
public void select(int index);
public void select(String itemName);
```

These methods serve the obvious purposes (noting that an item can be referred to either by its position in the list or by the actual text of the item).

For example, the following code will create an object of type `Choice` that contains the options Red, Blue, Green, and Black:

```
Choice colorChoice = new Choice();
colorChoice.add("Red");
colorChoice.add("Blue");
colorChoice.add("Green");
colorChoice.add("Black");
```

The most common way to use a `Choice` menu is to call its `getSelectedIndex()` method when you have a need to know which item in the menu is currently selected. However, like `Checkboxes`, `Choice` components generate `ItemEvents`. You can register an `itemListener` with the `Choice` component if you want to respond to such events when they occur.

The List Class

An object of type `List` is very similar to a `Choice` object. However, a `List` is presented on the screen as a scrolling list of items. Furthermore, you can create a `List` in which it is possible for the user to select more than one item in the list at the same time. The constructor for a `List` object takes the form

```
List(int itemsVisible, boolean multipleSelectionsAllowed);
```

The first parameter tells how many items are visible in the list at one time; if the list contains more than this number of items, then the user can use a scroll bar to scroll through the list. The second parameter determines whether or not the user can select more than one item in the list at the same time. If you leave out the second parameter, multiple selections are not allowed.

The `List` class includes the `add(String)` method to add an item to the end of the list. You can also add an item in a specified position in the list using the `add(String,int)` method. (Items are numbered starting from zero.) The `getItemCount()` method returns an `int` giving the number of items in the list. The method `remove(int)` deletes the item at a specified position, while `remove(String)` removes a specified item. The `select(int)` method can be used to select the item at the specified position in the list, and `deselect(int)` unselects the item.

If exactly one item in the list is selected, then the method `getSelectedIndex()` will return the index of that item, and `getSelectedItem()` will return the item itself. However, if no items are selected or if more than one item is selected, then `getSelectedIndex()` will return `-1`, and `getSelectedItem()` will return `null`. In that case, you can use the methods `getSelectedItems()` and `getSelectedIndexes()` to determine which items are selected. (However, these two methods return "arrays", which I will not cover until [Chapter 8](#).)

A `List` generates events of type `ItemEvent`, and you can register an `ItemListener` with a `List` if you want. An `ItemEvent` is generated whenever an item is selected. In the case of a list that allows multiple selections, an `ItemEvent` is also generated whenever an item is deselected. A `List` also generates an event of type `ActionEvent` when the user double-clicks on an item. The action command associated with such an `ActionEvent` is the text of the item on which the user double-clicked.

The TextField and TextArea Classes

`TextFields` and `TextAreas` are boxes where the user can type in and edit text. The difference between them is that a `TextField` contains a single line of editable text, while a `TextArea` displays multiple lines and might include scroll bars that the user can use to scroll through the entire contents of the `TextArea`. (It is also possible to set a `TextField` or `TextArea` to be read-only so that the user can read the text that it contains but cannot edit the text.)

Both `TextField` and `TextArea` are subclasses of `TextComponent`, which is itself a subclass of `Component`. The `TextComponent` class supports the idea of a **selection**. A selection is a subset of the characters in the `TextComponent`, 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 `TextComponent` include the following. They can, of course, be used for both `TextFields` and `TextAreas`.

```
public void setText(String newText); // substitute newText
                                   // for current contents
public String getText(); // return a copy of the current contents
public String getSelectedText(); // return the selected text
public select(int start, int end); // change the selected range;
    // characters in the range start <= pos < end are
    // selected; characters are numbered starting from zero
public int getSelectionStart(); // get starting point of selection
public int getSelectionEnd(); // get end point of selection
public void setEditable(boolean canBeEdited);
    // specify whether or not the text in the component
    // can be edited by the user
```

The constructor for a `TextField` takes the form

```
TextField(int columns);
```

where `columns` specifies the number of characters that should be visible in the text field. This is used to determine the 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 want the text field to expand to the maximum size available. In that case, you can use the constructor `TextField()`, with no parameters. You can also use the following constructors, which specify the initial contents of the text field:

```
TextField(String contents);
TextField(String contents, int columns);
```

The constructors for a `TextArea` are

```
TextArea();
TextArea(int lines, int columns);
TextArea(String contents);
TextArea(String contents, int lines, int columns);
```

The parameter `lines` specifies how many lines of text are visible in the text area. This determines the height of the text area. (The text area can actually contain any number of lines; a scroll bar is used to reveal lines that are not currently visible.) It is common to use a `TextArea` 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.

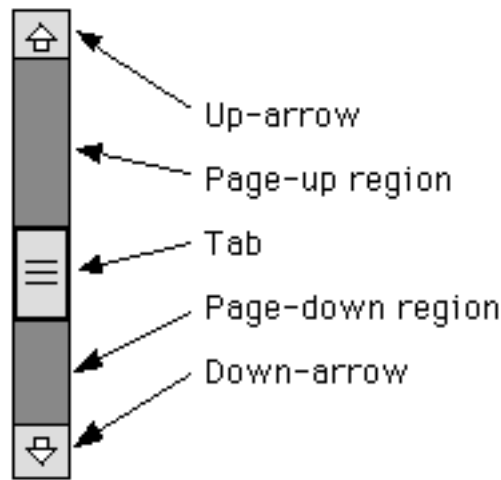
The `TextArea` class adds a few useful procedures to those inherited from `TextComponent`:

```
public 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
public void insert(String text, int pos);
    // insert the text, starting at specified position
public 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
```

A `TextField` generates an `ActionEvent` when the user presses return while typing in the `TextField`. The `TextField` class includes an `addActionListener()` method that can be used to register a listener with a `TextField`. In the `actionPerformed()` method, the `evt.getActionCommand()` method will return a copy of the text from the `TextField`. `TextAreas` do not generate action events.

The ScrollBar Class

Finally, we come to the `ScrollBar` class. A `ScrollBar` allows the user to select an integer value from a range of values. 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 scroll bar. On some platforms, 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 scroll bar 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)**.

These values can be specified when the scroll bar is created. The constructor takes the form

```
Scrollbar(int orientation, int value, int visible, int min, int max);
```

The **orientation**, which specifies whether the scroll bar is horizontal or vertical, must be one of the constants `Scrollbar.HORIZONTAL` or `Scrollbar.VERTICAL`. The **value** must be between **min** and **(max - visible)**. You can leave out all the **int** parameters to get a scroll bar with default values. You can set the **value** of the scroll bar at any time with the method `setValue(int)`. If you want to set any of the other parameters, you have to set them all, using

```
public 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 remaining question is, 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:

```
public void setUnitIncrement(int unitIncrement);
public void setBlockIncrement(int blockIncrement);
```

Let's look at an example. Suppose that you want to use a very large canvas, which is too large to fit on the screen. You might decide to display only part of the canvas and to provide scroll bars to allow the user to scroll through the entire canvas. Let's say that the actual canvas is 1000 by 1000 pixels, and that you will

show a 200-by-200 region of the canvas 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 might make more sense, 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. Most likely, that would be done by listening for `ComponentEvents` and defining a `componentResized()` method to adjust the values of the scroll bars.)

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.

Pop-up Menus

Pop-up menus differ from all the components discussed above because they are not components and they are not usually visible. The user calls up a pop-up menu by performing some platform-dependent action with the mouse. For example, this might mean clicking with the right mouse button or middle mouse button, or clicking the mouse while holding down the control key. On my Windows computer with a two-button mouse, I have to press both mouse buttons at the same time to call up a pop-up menu.

Here is an applet that uses a pop-up menu. It's an improved version of an applet you saw in [Section 5.4](#). You can place shapes on the drawing area by clicking on the buttons. Once a shape is on the drawing area, you can drag it around. A pop-up menu will appear if you click your mouse on a shape, using the appropriate action for calling up a pop-up menu on your platform. The menu includes commands for changing the color and size of the shape, for deleting the shape, and for moving it to the front of all the other shapes. If you select one of the commands, the menu disappears and the command is carried out. Try it:

Sorry, but your browser
doesn't support Java.

It's not much more difficult to use pop-up menus in a program than it is to use the above components. A pop-up menu generates an `ActionEvent` when the user selects a command from the menu. The action command associated with that event is the label of the menu item that was selected. You can register an `ActionListener` with the menu to listen for commands from the menu.

A pop-up menu is an object belonging to the class `PopupMenu`. A newly created pop-up menu is empty. Items can be added to the menu with its `add(String)` method. A separator line can be added with the `addSeparator()` method. (There's a lot more you can do with menu items, if you want to look it up.) If `pmenu` is a pop-up menu, it can be added to a component, `comp`, by calling `comp.add(pmenu)`.

However, this does not make the menu appear on the screen. To make a menu appear, a program has to call `pmenu.show(comp,x,y)`. The pop-up menu appears with its upper-left corner at the point (x,y), where the coordinates are given in `comp`'s coordinate system.

The only other question is when to show the menu. It should be shown when the user performs the platform-dependent mouse action that is used for calling up pop-up menus -- assuming that the action is performed in a context where popping up the menu makes sense. (In the above applet, for example, that means only when the user is clicking on a shape). To hear such actions, you have to listen for mouse events from the component. A `MouseEvent`, `evt`, has a boolean-valued method, `evt.isPopupTrigger()` that you can call to determine whether the user is trying to pop up a menu. This could theoretically occur in either the `mousePressed` or in the `mouseReleased` method, so you should test for the pop-up trigger in both of these methods. The `mousePressed` method might look something like this (`mouseReleased` would be similar):

```
public void mousePressed(MouseEvent evt) {
    if (evt.isPopupTrigger()) {
        int x = evt.getX();
        int y = evt.getY();
        ... // Maybe test if it makes sense to show the menu here.
        pmenu.show(this,x,y); // Assume that "this" is a component
                               // that is listening for its own
                               // mouse events and that pmenu is
                               // the pop-up menu that has been
                               // added to that component.
    }
    else {
        ...// handle a normal mouse click
    }
}
```

Note that this method just makes the menu appear on the screen. Any command generated by that menu must be handled elsewhere, in an `actionPerformed` method. The source code for the above applet can be found in the file [ShapeDrawWithMenu.java](#). Look there for a realistic example of using pop-up menus.

[[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.

As a first example, let's look at a simple calculator applet. This example demonstrates typical uses of `TextFields`, `Buttons`, and `Labels`, 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 `Label` 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 `Label`.

Sorry, but your browser
doesn't support Java.

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 `TextField` components to get the user's input. The `TextFields` 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 `TextField`: 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 text from the input box is used in an `actionPerformed()` method, when the user clicks on one of the buttons. When a component is created in one method and used in another, we need an instance variable to refer to it. In this case, I use two instance variables, `xInput` and `yInput`, of type `TextField` to refer to the input boxes. The `Label` that is used to display the result is treated similarly: A `Label` is created and added to the applet in the `init()` method. When an answer is computed in the `actionPerformed` method, the `Label`'s `setText()` method is used to display the answer in the label. I use an instance variable named `answer`, of type `Label`, to refer to the label.

The applet also has four `Buttons` and two more `Labels`. (The two extra labels display the strings "x =" and "y =".) I don't use 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 `Label`, `answer`. The other three rows each contain several components. Each of these rows is occupied by a `Panel` that has its own layout manager. The row that contains the four buttons is a `Panel` that uses a `GridLayout` with one row and four columns. The `Panels` that contain the input boxes use `BorderLayouts`. The input box occupies the `Center` position of the `BorderLayout`, with a `Label` 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() {
```

```

setBackground(Color.lightGray);

/* Create the input boxes, and make sure that the background
   color is white. (On some platforms, it is automatically.) */

xInput = new TextField("0");
xInput.setBackground(Color.white);
yInput = new TextField("0");
yInput.setBackground(Color.white);

/* Create panels to hold the input boxes and labels "x = " and
   "y = ". By using a BorderLayout with the TextField in the
   Center position, the TextField will take up all the space
   left after the label is given its preferred size. */

Panel xPanel = new Panel();
xPanel.setLayout(new BorderLayout(2,2));
xPanel.add( new Label(" x = "), BorderLayout.WEST );
xPanel.add(xInput, BorderLayout.CENTER);

Panel yPanel = new Panel();
yPanel.setLayout(new BorderLayout(2,2));
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. */

Panel buttonPanel = new Panel();
buttonPanel.setLayout(new GridLayout(1,4));

Button plus = new Button("+");
plus.addActionListener(this); // Applet will listen for
buttonPanel.add(plus);         // events from the buttons.

Button minus = new Button("-");
minus.addActionListener(this);
buttonPanel.add(minus);

Button times = new Button("*");
times.addActionListener(this);
buttonPanel.add(times);

Button divide = new Button("/");
divide.addActionListener(this);
buttonPanel.add(divide);

/* Create the label for displaying the answer (in red). */

answer = new Label("x + y = 0", Label.CENTER);
answer.setForeground(Color.red);

/* Set up the layout for the applet, using a GridLayout,
   and add all the components that have been created. */

```

```

        setLayout(new GridLayout(4,1,2,2));
        add(xPanel); // (Calls the add() method of the applet itself.)
        add(yPanel);
        add(buttonPanel);
        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 label, answer
else if the command is "-"
    subtract the numbers and display the result in the label, answer
else if the command is "*"
    multiply the numbers and display the result in the label, answer
else if the command is "/"
    divide the numbers and display the result in the label, answer

```

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. It's possible to convert a `String` to a number. Unfortunately, Java doesn't make it easy. The following code will get the contents of the input box, `xInput`, and convert it to a value of type `double`:

```

String xStr = xInput.getText();
Double d = new Double(xStr);
x = d.doubleValue(); // where x is a variable of type double.

```

An object belonging to the standard class, `Double`, contains a value of type `double`. (`Double` is called a **wrapper class**. An object of type `Double` is a wrapper for a value of type `double`. This class exists because a value of type `double` is not an object, and some contexts require objects. If you want to use a `double` value in such a context, you have to wrap it in an object of type `Double`.) The constructor "`new Double(xStr)`" converts `xStr` to a `double` value and puts it in an object of type `Double`. The function `d.doubleValue()` gets the numerical value from that object. This is really unnecessarily complicated, isn't it? (Things are a little easier for integers. If you want to convert a `String`, `str`, to an `int` value, you can say "`int N = Integer.parseInt(str)`". `Integer` is the wrapper class for values of type `int`, and `parseInt()` is a static method in that class. For some reason, there is no `parseDouble()` method in the `Double` class.)

The complications are not over. If the string `xStr` does not contain a legal number, then the constructor "`new Double(xStr)`" generates an error. It is good style to catch this error and display an error message to the user. Catching the error requires the `try...catch` statement, which will be covered in [Chapter 9](#). In the meantime, you can see how it's done in the `actionPerformed()` method from the applet. Aside from the difficulty of getting numerical values from the input boxes, the method is straightforward:

```

public void actionPerformed(ActionEvent evt) {

    double x, y; // The numbers from the input boxes.

    /* Get a number from the xInput TextField. 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();
        Double d = new Double(xStr);
        x = d.doubleValue();
    }
    catch (NumberFormatException e) {
        answer.setText("Illegal data for x.");
        return; // Break out of the actionPerformed method.
    }

    /* Get a number from yInput in the same way. */

    try {
        String yStr = yInput.getText();
        Double d = new Double(yStr);
        y = d.doubleValue();
    }
    catch (NumberFormatException e) {
        answer.setText("Illegal data for y.");
        return; // Break out of the actionPerformed method.
    }

    /* 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 this applet can be found in the file [SimpleCalculator.java](#). (It contains very

little in addition to the two methods shown above.)

As a second example, let's look more briefly at another applet. In this example, the user manipulates three scrollbars to set the red, green, and blue levels of a color. The value of each color level is displayed in a label, and the color itself is displayed in a large rectangle:

Sorry, but your browser
doesn't support Java.

The layout manager for the applet is a `GridLayout` with one row and three columns. The first column contains a `Panel` that uses another `GridLayout`, which contains three `Scrollbar`s. The second also uses a `GridLayout` to contain three `Labels`. The third column contains the colored rectangle. The component in this column is a `Canvas`. The displayed color is the background color of the canvas. When the user changes the color, the background color of the canvas is changed and the canvas is repainted. This is one of the few cases where an object of type `Canvas` is used, rather than an object belonging to a subclass of the `Canvas` class.

When the user changes the value on a scrollbar, an event of type `AdjustmentEvent` is generated. In order to respond to such events, the applet implements the `AdjustmentListener` interface, which specifies the method `"public void adjustmentValueChanged(AdjustmentEvent evt)"`. The applet registers itself to listen for adjustment events from each scrollbar. The applet has instance variables to refer to the scrollbars, the labels, and the canvas. Let's look at the code from the `init()` method for setting up one of the scrollbars, `redScroll`:

```
redScroll = new Scrollbar(Scrollbar.HORIZONTAL, 0, 10, 0, 265);
redScroll.setBackground(Color.lightGray);
redScroll.addAdjustmentListener(this);
```

The first line constructs a horizontal scrollbar whose initial value is 0. The entire length of the scroll bar represents numbers between 0 and 265, as specified by the last two parameters in the constructor. However, the tab of the scrollbar takes up 10 units, as specified in the third parameter, so the value of the scrollbar is actually restricted to the range from 0 to 255. These are the possible values of a color level. In the second line, the background color of the scrollbar is set. On some platforms, all scrollbars are the same color and this command is ignored. On other platforms, every component inherits its color from its container, and this can look unattractive. The third line registers the applet ("this") to listen for adjustment events from the scrollbar.

In the `adjustmentValueChanged()` method, the applet must respond to the fact that the user has changed the value of one of the scroll bars. The response is to read the values of all the scrollbars, set the labels to display those values, and change the color displayed by the canvas. (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 adjustmentValueChanged(AdjustmentEvent evt) {
    int r = redScroll.getValue();
    int g = greenScroll.getValue();
    int b = blueScroll.getValue();
    redLabel.setText(" R = " + r);
    greenLabel.setText(" G = " + g);
    blueLabel.setText(" B = " + b);
    colorCanvas.setBackground(new Color(r,g,b));
    colorCanvas.repaint(); // Redraw the canvas in its new color.
} // end adjustmentValueChanged
```

The complete source code can be found in the file [RGBColorChooser.java](#).

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 `Canvas` class. A `Canvas` 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 display can be done with a `Label`, and I can define my `StopWatch` component as a subclass of the `Label` 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 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.*;
import java.awt.event.*;

public class StopWatch extends Label 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. First, call the constructor from
        // the superclass, Label. Then, set the component to
        // listen for mouse clicks on itself.
        super(" Click to start timer. ", Label.CENTER);
        addMouseListener(this);
    }

    public void mousePressed(MouseEvent evt) {
        // React when user presses the mouse.
        // Start the timer or stop it if it is already running.
        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 `Label`, you can do anything with a `StopWatch` that you can do with a `Label`. You can add it to a container. You can set its font, foreground color, and background color. You can even set the text that it displays (although this would interfere with its stopwatch function).

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 `Label` 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 `Canvas`. It has a constructor that specifies the text to be displayed and a `setText()` method that changes the displayed text. The `paint()` 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 canvas. The text is drawn to the canvas, in the usual way. Then the canvas is copied to the screen with the command:

```

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.*;

public class MirrorLabel extends Canvas {

    // 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 invalidate so that
        // its size will be computed if its container is validated.
        this.text = text;
        invalidate();
        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 canvas holding
                       // the non-reversed text.
    private int widthOfOSC, heightOfOSC; // Size of off-screen canvas.

    public void update(Graphics g) {
        // Redefine update so that it calls paint without erasing.
        paint(g);
    }

    public void paint(Graphics g) {
        // The paint method makes a new OSC, if necessary. It writes
        // a non-reversed copy of the string to the OSC, then reverses
        // the OSC as it copies it to the screen.
        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, this);
    }

    public Dimension getMinimumSize() {
        return getPreferredSize();
    }

    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 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. The size is not always respected. For example, in a GridLayout, every component is sized to fit the available space. However, in some cases the preferred size is essential. For example, the height of a component in the North or South position of a BorderLayout is given by the component's preferred size. Java's standard GUI components already define a getPreferredSize() method. But when you define a component as a subclass of Canvas, you should include a preferred size method. (You are also supposed to include a getMinimumSize() method, but I don't know of any case where the minimum size is actually respected.)

Here is an applet that demonstrates a MirrorLabel and a Stopwatch component. The applet uses a FlowLayout, so the components are not arranged very neatly. The applet also contains two buttons, which are there to illustrate another fine point of programming with components. (Don't forget to try the Stopwatch!)

Sorry, but your browser
doesn't support Java.

If you click the button labeled "Change Text in this Applet", the text in all the components will be changed. However, you will notice that the components don't change size. That won't happen until you click the other button. Here's how it works: When you click the button labeled "Validate" or "Do Validation", the applet's validate() method is called. The validate() method computes new sizes for components in the applet and lays out the applet again. However, it only computes a new size for a component if that component has been declared to be "invalid." This is done by calling the component's invalidate() method. If you look at the source code for MirrorLabel, you'll see that the setText() method calls invalidate(). This means that when the text in a MirrorLabel is changed, the MirrorLabel is marked as being invalid. The next time the applet is validated, the size of the MirrorLabel will be changed. The situation for Java's standard components is not completely clear. On my computer, the Buttons change size, but the Stopwatch does not. The best programming style requires that when you make a change to a component that might require a change in size, you should call the component's invalidate() method and call the validate() method of the container that contains the component. (In practice, I rarely do this, and only after I see a problem when I run the program.)

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. If comp is a component, then the statement

```
comp.setBounds(x, y, width, height);
```

puts the top left corner 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 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 canvas that displays a checkerboard pattern. This applet doesn't do anything useful. The buttons just change the text in the label.

Sorry, but your browser
doesn't support Java.

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() {

    setLayout(null); // I will do the layout myself!

    setBackground(new Color(0,150,0)); // Dark green background.

    /* Create the components and add them to the applet.  If you
       don't add them to the applet, they won't appear, even if
       you set their bounds! */

    board = new SimpleCheckerboardCanvas();
    add(board);

    newGameButton = new Button("New Game");
    newGameButton.setBackground(Color.lightGray);
    newGameButton.addActionListener(this);
    add(newGameButton);

    resignButton = new Button("Resign");
    resignButton.setBackground(Color.lightGray);
    resignButton.addActionListener(this);
    add(resignButton);

    message = new Label("Click \"New Game\" to begin a game.",
                        Label.CENTER);
    message.setForeground(Color.green);
    message.setFont(new Font("Serif", Font.BOLD, 14));
    add(message);

    /* Set the position and size of each component by calling
       its setBounds() method.  It is assumed that this applet
       is 330 pixels wide and 240 pixels high.  */

    board.setBounds(20,20,164,164);
    newGameButton.setBounds(210, 60, 100, 30);
    resignButton.setBounds(210, 120, 100, 30);
    message.setBounds(0, 200, 330, 30);
}
```

It's 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. One way to do this is by listening for `ComponentEvents` from the container and doing the computations in the `componentResized()` method. 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.

Section 7.5

Threads, Synchronization, and Animation

MUST AN APPLET BE COMPLETELY DEPENDENT on events sent from outside to get anything done? Can't an applet do something on its own initiative and on its own schedule? Something more like a traditional program that just executes a sequence of instructions from beginning to end?

The answer is yes. The applet can create a **thread**. A thread represents a thread of control, which independently executes a sequence of instructions from beginning to end. Several threads can exist and run at the same time. An applet -- or, indeed, any Java program -- can create one or more threads and start them running. Each of the threads acts as a little program. The separate threads run independently but they can communicate with each other by sharing variables.

One common use of threads is to do animation. A thread runs continuously while an applet is displayed. Several times a second, the thread changes the applet's display. If the changes are frequent enough, and the changes small enough, the viewer will perceive continuous motion.

Programming with threads is called **parallel programming** because several threads can run in parallel. Parallel programming can get tricky when several threads are all using a **shared resource**. An example of a shared resource is the screen. If several threads try to draw to the screen at the same time, it's possible for the contents of the screen to become corrupted. Instance variables can also be shared resources. In order to avoid problems with shared resources, access to shared resources must be **synchronized** in some way. Java defines a fairly natural and easy way of doing such synchronization. I will discuss it below.

Note that on most computers, you can't literally have two threads running "at the same time," since there is only one processor, which can only do one thing at a time. Only computers with more than one processor can literally do more than one thing at a time. However, this does **not solve the synchronization problem on single-processor computers!** A single-processor computer simulates multiprocessing by switching rapidly from thread to thread. Without proper synchronization, a thread can be interrupted at any time to let another thread run, and this can cause problems. Suppose, for example that a thread reads the value of an important variable just before it is interrupted. After the thread resumes, it makes a decision based on the value it just read. The problem is that, without proper synchronization, some other thread might have butted in and changed the value of the variable in the meantime! The decision that the thread makes is based on a value that is no longer necessarily valid. Synchronization can be used to make sure this doesn't happen.

Even if an applet creates only one thread, there will still be two threads running in parallel, since there is always a user interface thread that monitors user actions and feeds events to the applet. What happens if the thread is trying to draw something at the same time that the user changes the size of the applet? What if one thread is drawing to an off-screen image at the same time another thread is copying that image to the screen? These are synchronization problems. So, even in the simple case of an applet that creates a single thread, synchronization can be an issue.

In Java, a thread is just an object of type `Thread`. The class `Thread` is defined in the standard package `java.lang`. A thread object must have a subroutine to execute. That subroutine is always named `run()`, but there are two different places where the `run()` method might be located, depending on how the thread is programmed.

The `Thread` class itself defines a `run()` method, which doesn't do anything. One way to make a useful thread is to define a subclass of `Thread` and override the `run()` method in your subclass to make it do something useful.

A second way to program a thread -- and the only one I will use for the time being -- is to create a class that

implements the interface called `Runnable`. This interface defines one method, "`public void run()`". (To implement this interface means to declare that the class "implements `Runnable`" and to define the method "`public void run()`" in the class.) A thread can be constructed from an object of type `Runnable`. When such a thread is run, it executes the `run()` method from the `Runnable` object. I'll give an example of all this in just a moment. The advantage of defining a thread in this way is that the `run()` method has access to all the instance variables of the object, so that the thread will be able to read and change the values of those variables. (A disadvantage is that throwing a `run()` method into a class can completely muddle the clear division of responsibilities that should be the hallmark of object-oriented programming. The `run()` method is logically part of the `Thread` object, but it is physically part of the `Runnable` object. Keep this in mind: It's best to think of the `run()` method as a separate entity.)

Suppose that `runnableObject` is an object that implements `Runnable`. And suppose that `runner` is a variable of type `Thread`. Then the statement

```
runner = new Thread(runnableObject);
```

creates a thread that can execute the `run()` method of `runnableObject`. However, the thread does not automatically start running. To get it to run, you have to call its `start()` method:

```
runner.start();
```

The thread will then begin executing the `run()` method. At the same time, the rest of your program continues to execute. The thread continues until it reaches the end of the `run()` method. At that point it "dies" and cannot be restarted or reused. If you want to execute the same `run()` method again, you have to create a new thread to do it. You can check whether a thread is still alive by calling its `isAlive()` method, which returns a boolean value:

```
if (runner.isAlive()) . . .
```

(Note: it is possible to kill a thread before it finishes normally by calling its `stop()` method. However, use of this method is discouraged since it is error-prone, and I will avoid it entirely.)

As we work through a few examples in the rest of this section, I'll be introducing several important new ideas. One of the ideas has to do with managing the state of an applet. The state of an applet consists of its instance variables. The state determines how the applet will respond when its methods are called. Managing the state means determining how and when the state should change. In many cases, it also means making the state of the applet apparent to the user, so the user isn't surprised or frustrated by the applet's behavior. One way to make the state of the applet apparent is to disable a button whenever clicking on that button would make no sense. Recall that a button, `btn`, can be disabled with the command "`btn.setEnabled(false);`", and it can be enabled with "`btn.setEnabled(true);`".

Let's look at the first example. When you click on the button in the following applet, a thread is created that blinks the color of the message from red to green and back again several times. Note that the button is disabled while the message is blinking:

Sorry, but your browser
doesn't support Java.

The source code for this applet begins with the lines:

```
public class BlinkingHelloWorld1 extends Applet
    implements ActionListener, Runnable {

    ColoredHelloWorldCanvas canvas; // Canvas that displays the message
```

```
Button blinkBtn; // The "Blink at Me" button.
```

```
Thread blinker; // A thread that cycles the message colors.
```

In this example, the applet class itself implements `Runnable`. (There is nothing special about applets in this regard. Any class can implement `Runnable`. It might be useful, for example, to have a `Runnable` canvas.) Later in the source code, the thread will be created with the command `"blinker = new Thread(this);"`, where `this` refers to the applet object itself. This means that the thread we create will execute whatever `run()` method is defined in the `BlinkingHelloWorld1` class. Let's think about what we want this thread to do. (This is just like designing a small program.) We want the thread to blink the color of the message. We'll also give the thread the responsibility of disabling and enabling the button, since that way we can be sure that the button will be disabled just as long as the message is blinking. A pseudocode version of the `run` method would be:

```
disable the blinkBtn
set the text color to green
pause for a while
set the text color to red
pause for a while
set the text color to green
pause for a while
set the text color to red
pause for a while
set the text color to green
pause for a while
set the text color to red
enable the blinkBtn
```

The pauses are necessary since otherwise the blinking would go by much too fast to see. Unfortunately, for technical reasons, getting a thread to pause requires a `try...catch` statement, which will not be covered until [Chapter 9](#). For the time being, I will just give you a subroutine that can be called by a thread to insert a pause in its execution:

```
void delay(int milliseconds) {
    // Pause for the specified number of milliseconds,
    // where 1000 milliseconds equal one second.
    try {
        Thread.sleep(milliseconds);
    }
    catch (InterruptedException e) {
    }
}
```

Using this subroutine and a `setTextColor()` routine from the `ColoredHelloWorldCanvas` class, the applet's `run()` method becomes:

```
public void run() {
    blinkBtn.setEnabled(false);
    canvas.setTextColor(Color.green);
    delay(300);
    canvas.setTextColor(Color.red);
    delay(300);
    canvas.setTextColor(Color.green);
    delay(300);
    canvas.setTextColor(Color.red);
    delay(300);
    canvas.setTextColor(Color.green);
}
```



```

        delay(300);
        canvas.setTextColors(Color.red);
        blinkBtn.setEnabled(true);
    }

```

The `blinker` thread, which executes this `run` method, has to be created and started when the user clicks the "Blink at Me!" button. This is done in the applet's `actionPerformed()` method, which is called when the user clicks the button:

```

public void actionPerformed(ActionEvent evt) {
    if ( blinker == null || (blinker.isAlive() == false) ) {
        blinker = new Thread(this);
        blinker.start();
    }
}

```

This routine creates and starts a thread. That thread executes the above `run()` method, which blinks the text several times. When the `run` method ends, the thread dies.

I only want one of these threads to be running at a time. To be sure of this, before creating the new thread, I test whether another thread already exists and is running. Since the button is disabled while a thread is running, this test might seem unnecessary. However, it's usually better to test a condition rather than just assume it's true. And in this case, it's just possible that the user might manage to click the button a second time before the first thread gets started.

The complete source code for this example can be found in the file [BlinkingHelloWorld1.java](#).

In the previous example, the thread that was created ran for only a short time, until it had run through all the instructions in its `run()` method. In many cases, though, we want a thread to run over an extended period. Often a thread that is created by an applet should run as long as the applet itself exists. Before working with such threads, though, you need to know more about an applet's **life cycle**.

An applet, just like any other object, has a "life cycle." It is created, it exists for a time, and it is destroyed. The `Applet` class defines an `init()` method, which is called just after the applet is created. There are other methods in the `Applet` class which are called at other points in an applet's life cycle. A programmer can provide definitions of these methods if there are tasks that the programmer would like to have executed at those points in the applet's life cycle.

Just before an applet object is destroyed, the method "`public void destroy()`" is called to give the applet a chance to clean up before it ceases to exist. Because of Java's automatic garbage collection, a lot of cleanup is done automatically. There are a few cases, however, where `destroy()` might be useful. In particular, if the applet has created any threads, those threads should almost certainly be stopped before the applet is destroyed. The `destroy()` method is a natural place to do this. As another example, it is possible for an applet to create a separate window on the screen. The applet's `destroy` method could close such a window so it doesn't hang around after the applet no longer exists.

An applet also has methods "`public void start()`" and "`public void stop()`". These play similar roles to `init()` and `destroy()`. However, while `init()` and `destroy()` are each called exactly once during the life cycle of an applet, `start()` and `stop()` can be called many times. The `start()` method is definitely called by the system at least once when the applet object is first created, just after the `init()` method is called. The `stop()` method will definitely be called at least once, before the applet is destroyed. In addition to these calls, the system can choose to stop the applet by calling its `stop()` method at any time and then later restart it by calling its `start()` method again. For example, a Web browser will typically stop an applet if the user leaves the page on which the applet is displayed, and will restart it if the user returns to that page. An applet that has been stopped will not receive any other events until it has been restarted.

It is not always clear what initialization should be done in `init()` and what should be done in `start()`. Things that only need to be done once should ordinarily be done in `init()`. If the applet creates a separate thread to carry out some task, it is reasonable to start that thread in the `start()` method and stop it in the `stop()` method. If the applet uses a large amount of some computer resource such as memory, it might be reasonable for it to allocate that resource in its `start()` method and release it in its `stop()` method, so that the applet will not be hogging resources when it isn't even running. On the other hand, sometimes this is impossible because the applet needs to retain the resource for its entire lifetime.

The next example creates a thread that runs until the user clicks on a button, or until the applet itself is stopped:

Sorry, but your browser
doesn't support Java.

When you click on the "Blink!" button, the message starts cycling between two colors. The name of the "Blink!" button changes to "Stop!". Clicking on the "Stop!" button will stop the blinking. (The command that changes the name of the button, `blinkBtn`, to "Stop!" is `blinkBtn.setLabel("Stop!")`. This is another case of changing the appearance of the applet when it changes state in order to help the user understand what is going on.) The other two buttons can be used at any time to set the colors that are used for the blinking message. Recall that the blinking is handled by one thread while the button events are handled by a separate user interface thread, so in this example you really do get to see two threads operating in parallel.

The `run()` method for this example has a `while` loop that makes it blink the text over and over. The interesting question is how to get the loop to end and the thread to stop running when the user clicks the "Stop!" button. The answer is to use a shared instance variable for communication between the thread and the applet. The thread tests the value of the variable and keeps running as long as the variable has a certain value. When the user clicks on the "Stop!" button, the applet changes the value of the variable. The thread sees this change and responds by exiting from its `run()` method and dying.

I tend to use a variable named `status` for this type of communication. It's a good idea, for the sake of readability, to use named constants as the possible values of `status`. For this example, the status information that I need is whether the thread should continue or should end. I use constants named `GO` and `TERMINATE` to represent these two possibilities. The thread continues running as long as the value of `status` is `GO`. It ends when the value of `status` changes to `TERMINATE`. To implement this, the applet includes the following instance variables:

```
private final static int GO = 0,           // For use as values of status.
                        TERMINATE = 1;

private volatile int status; // This is used for communication between
                             // the applet and the thread. The value is
                             // set by the applet to tell the thread what
                             // to do. When the applet wants the thread
                             // to terminate, it sets the value of status
                             // to TERMINATE.
```

(The word `volatile` is a new modifier that has to do with communication between threads. It should be used on a variable whose value is set by one thread and read by another. The somewhat technical reason is this: If a variable is not declared to be `volatile`, then on some computers, when one thread changes the value of the variable, other threads might not immediately see the new value. This is another type of synchronization problem. Later in this section, I'll mention "synchronized" methods and statements. Variables that are accessed only in synchronized methods and statements don't have to be declared `volatile`.)

The idea is for the thread to run just so long as the value of `status` is `GO`. The thread's `run()` method tests the value of `status` in a `while` loop, which continues only so long as `status == GO`. Note that the value of `status` must be set equal to `GO` before the thread is started, since otherwise the `run()` method would finish immediately. When the user clicks on the "Blink!" button, the following commands are executed to start the thread:

```
runner = new Thread(this);
status = GO;
runner.start();
```

In addition to blinking the text, the thread is also responsible for changing the label on the button. Here is the entire `run()` method for the thread:

```
public void run() {
    blinkBtn.setLabel("Stop!");
    while (status == GO) {
        waitDelay(300);
        changeColor();
    }
    blinkBtn.setLabel("Blink!");
}
```

Here, the `waitDelay()` method imposes a delay of 300 milliseconds, while `changeColor()` changes the color of the displayed message. Both these methods are defined elsewhere in the applet.

The thread is started and stopped in the `actionPerformed()` method, in response to the user clicking on the Blink/Stop button. When the user clicks "Stop!", the value of `status` is set to `TERMINATE`. The thread sees this value and stops. But we have to be careful. What if the user never clicks on "Stop!"? We should be careful not to leave the thread running after the applet is destroyed. An applet that creates threads that might otherwise run forever should make sure to terminate them, either in its `stop()` method or in its `destroy()` method. In this example, I use the `stop()` method to stop the thread by setting `status` to `TERMINATE`. The `stop()` method will be called by the system if you close the window that displays this page or follow a link to another page (or, in Netscape, if you just resize the page). So, if you start the message blinking, go to another page, and then return to this one, the blinking should be stopped.

In the second blinking hello world applet, there are two reasons why the color of the message might change: because the `runner` thread changes it or because the user clicks on the "Red/Green" button or the "Black/Blue" button. These two reasons are handled by two different threads. The variables that record the current color are resources that are shared by two threads. When a thread is accessing a shared resource, it usually needs **exclusive access** to that resource, so that no other thread can butt in and access the resource at the same time. In Java, exclusive access is implemented using **synchronized methods** and **synchronized statements**. A method is declared to be synchronized by adding the `synchronized` modifier to its definition. Here, for example, is the method that is called by the `runner` thread to change the color of the message:

```
synchronized void changeColor() {
    // Change from first to second color or vice versa.
    if (showingFirstColor) {
        // Change to showing second color.
        showingFirstColor = false;
        if (useRedAndGreen)
            canvas.setTextColors(Color.green);
        else
            canvas.setTextColors(Color.blue);
    }
}
```

```

        else {
            // Change to showing first color.
            showingFirstColor = true;
            if (useRedAndGreen)
                canvas.setTextColor(Color.red);
            else
                canvas.setTextColor(Color.black);
        }
    } // end changeColor()

```

The other method that can change the color of the text is the `actionPerformed()` method, which is called by the user interface thread when the user clicks on one of the buttons. Like the `changeColor()` method, the `actionPerformed()` method is declared to be synchronized. This means that all the code that does color changes is contained inside synchronized methods. Therefore, only one of the color-change processes can be running at any given time, and there is no possibility of their interfering with each other.

Here, briefly, is how such synchronization is implemented in Java: Every object in Java has an associated **lock**. The lock can be "held" by a thread, but only one thread can hold the lock at a given time. The rule is that in order to execute a synchronized method in an object, a thread must obtain that object's lock. If the lock is already held by another thread, then the second thread must wait until the first thread releases the lock. If all access to a shared resource takes place inside methods that are synchronized on the same object, then each thread that accesses the resource has exclusive access.

When a resource is only used in part of a method, it's not necessary to make the entire method synchronized. A single statement can be synchronized. The synchronized statement takes the form

```

synchronized( object ) {
    statements
}

```

(The synchronized statement, for some unknown reason, requires the braces `{` and `}` even if they contain just a single statement.) To execute the **statements**, a thread must first obtain the lock belonging to the specified **object**. Often, you will say `"synchronized(this)"` to synchronize on the same object that contains the method. However, it is also possible to synchronize on another object's lock.

What could go wrong in the sample applet if the methods were not synchronized? For example: When the blinker thread executes, the statement

```

if (useRedAndGreen)
    canvas.setTextColor(Color.green);
else
    canvas.setTextColor(Color.blue);

```

it is possible that just after the thread tests the value of `useRedAndGreen`, the other thread butts in and changes the value of `useRedAndGreen`. Then the blinker thread resumes and changes the text color based on the value of `useRedAndGreen` that it saw. However, that value is no longer valid, and the thread changes the text to an incorrect color. The state of the applet has become inconsistent. The variables say the color should be one thing, but the actual color is different. It's not a big deal here, but there are cases where something like this -- even if it happens very, very rarely -- could be a really big deal.

The complete source code for this example is in the file [BlinkingHelloWorld2.java](#).

In the rest of this section, I will try to explain the most advanced aspect of synchronization, the `wait()` and `notify()` methods. These methods are defined in the `Object` class, and so they can be used with any object. In order to legally call an object's `wait()` method, a thread must hold that object's lock. So `wait()` is usually called only in synchronized methods and statements.

A thread calls an object's `wait()` method when it wants to wait for some event to occur. (While it is waiting, it releases its hold on the lock, so that other threads can run.) Some other thread must call the same object's `notify()` method when the event occurs. When an object's `notify()` method is called, any threads that are waiting on that object will wake up and can continue. Obviously, this requires close coordination between several threads, which means very careful programming.

If `notify()` is never called, it's possible that a waiting thread might wait forever. You can avoid this with good programming, but you can also put a time limit on how long the thread will wait. This is done by passing a parameter to the `wait()` method specifying the maximum number of milliseconds that the thread will wait. If `notify()` has not been called by the end of that time period, the thread will wake up anyway.

As with an earlier example on this page, use of the `wait()` method requires a `try...catch` statement. Here are two methods that can be called to wait indefinitely or for a specified time period:

```
synchronized void waitDelay(int milliseconds) {
    // Pause for the specified number of milliseconds OR
    // until the notify() method is called by some other thread.
    try {
        wait(milliseconds);
    }
    catch (InterruptedException e) {
    }
}

synchronized void waitDelay() {
    // Pause until the notify() method is called
    // by some other thread.
    try {
        wait();
    }
    catch (InterruptedException e) {
    }
}
```

The first of these can be used as a kind of interruptable delay. A thread that calls it will pause for the specified number of milliseconds, unless a call to `notify()` occurs in the meantime. I use this `waitDelay()` method in the `run()` method of the `BlinkingHelloWorld2` applet. Whenever I set the status variable in that applet to `TERMINATE`, I call `notify()`. If the thread is in the middle of a `waitDelay()`, this will wake it up. Suppose you click on the "Stop!" button just after the runner thread calls `waitDelay(300)`. Because of the call to `notify()`, the runner thread wakes up and terminates immediately, instead of continuing to sleep for 300 milliseconds before terminating. This avoids a noticeable delay between the time you click on the button and the time that its name changes back to "Blink!". However, this is still a pretty trivial use of `wait()` and `notify()`. The final example in this section shows how to use them in a non-trivial way. It also provides an example of using an off-screen canvas to do smooth animation. Here's the last "Hello World" you'll see in these notes:

**Sorry, but your browser
doesn't support Java.**

The complete source code for this applet can be found in the file [ScrollingHelloWorld.java](#). I'll only look at a few aspects of it here.

The thread that animates this applet runs continually as long as this page is visible. The thread is created when the applet is first started, and it is stopped when the applet is destroyed. However, during periods when the applet is stopped, the thread is not actively running. It is just waiting to be notified when the

applet is restarted. This behavior is programmed using the applet's `wait()` and `notify()` methods.

The runner thread in this applet has three possible statuses: `GO` to tell it to run the animation; `TERMINATE` to tell it to terminate because the applet is about to be destroyed; and `SUSPEND` to tell it that the applet has been stopped, and that it should go to sleep and wait for the applet to be restarted. The applet's `start()` method sets the status to `GO`. If the thread does not yet exist, it is created and started. Otherwise, the existing thread is notified that the value of `status` has changed. This will wake up a thread that has been put to sleep by a previous call to `stop()`:

```
synchronized public void start() {
    // Called when the applet is being started or restarted.
    // Create a new thread or restart the existing thread.
    status = GO;
    if (runner == null || !runner.isAlive()) {
        // Thread doesn't yet exist or has died for some reason.
        runner = new Thread(this);
        runner.start();
    }
    else {
        // Another thread exists and is still alive, but
        notify(); // is presumably sleeping. Wake it up.
    }
}
```

The `stop()` and `destroy` methods merely change the value of `status` and notify the thread that the value has changed:

```
synchronized public void stop() {
    // Called when the applet is about to be stopped.
    // Suspend the thread.
    status = SUSPEND;
    notify();
}

synchronized public void destroy() {
    // Called when the applet is about to be permanently
    // destroyed. Stop the thread.
    status = TERMINATE;
    notify();
}
```

The `run()` method for this example uses the following `while` loop to run the animation:

```
while (status != TERMINATE) {
    synchronized(this) {
        while (status == SUSPEND)
            waitDelay();
    }
    if (status == GO)
        nextFrame();
    if (status == GO)
        waitDelay(250);
}
```

This loop is repeated until `status` becomes equal to `TERMINATE`. The `synchronized` statement in this loop,

```
synchronized(this) {
    while (status == SUSPEND)
        waitDelay();
}
```

```
}
```

causes the thread to pause as long as the `status` is `SUSPEND`. It's good practice here to use a `while` statement rather than an `if` statement, since in general `notify()` could be called for several different reasons. Just because `notify()` has been called, it doesn't necessarily mean that the value of `status` has changed from `SUSPEND` to something else.

You might wonder why this is synchronized. If it were not synchronized, the following sequence of events would be possible: (1) The runner thread finds that the value of `status` is `SUSPEND` and decides to call `waitDelay()`; (2) some other thread butts in, changes the value of `status` and calls `notify()`; (3) the runner thread resumes and calls `waitDelay()` after `notify()` has already been called. Then the `waitDelay()` will cause the thread to wait for a `notify()` that has already occurred and is not going to occur again. This would be a minor disaster: The animation will not properly restart, or the thread will not properly terminate. Again, in other circumstances, the disaster could be major.

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

Section 7.6

Nested Classes and Adapter 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** 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.

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`.

To understand non-static nested classes, you have to stretch your mind a bit. Non-static members of a class are not really part of the class itself. This is just as true for non-static nested classes 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**. For example, from outside the containing class, a non-static nested class has to be referred to as **`objectName.NestedClassName`**, rather than as **`ContainingClassName.NestedClassName`**. The non-static nested class cannot be used in the static methods of the containing class. It can, of course, be used in the non-static methods of the containing class -- and in that case, what's being used is really the copy of the nested class associated with the "this" object in that method, as if you had said "this.**`NestedClassName`**".

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. 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. Here is a simple applet that shows four randomly colored squares. Each square is a `Canvas` object. The applet runs a thread that every so often picks one of the squares and changes its color:

Sorry, but your browser
doesn't support Java.

The thread in this applet could have been programmed by declaring that the applet class "implements `Runnable`" and adding a `run()` method to the class. Instead of doing it that way, however, I decided to define a non-static nested class to represent the thread object. The nested class is a subclass of the `Thread` class, and it has its own `run()` method. Because of the nesting, the thread object has access to the instance variables of the applet object, so communication between the thread and the applet is no problem. Here's the full definition of the applet class.

```
public class RandomColorGrid extends Applet {

    ColorSwatchCanvas canvas0, canvas1, canvas2, canvas3;
    // Canvases to be displayed in applet. (The definition
    // of the ColorSwatchCanvas class will be given below.)

    Thread runner;    // A thread that randomly changes the color
                     // of a canvas every so often. Note that all
                     // synchronization in this applet is
                     // done on the Thread object.

    volatile int status; // Status variable for controlling the thread.

    final static int GO = 0,           // Possible values for status.
                   SUSPEND = 1,
                   TERMINATE = 2;

    class Runner extends Thread {

        // The runner for the RandomColorGrid canvas will be an
        // object belonging to this nested class.

        public void run() {
            // Run method picks a random canvas, changes it's color,
            // waits for a random time between 30 and 3029
        }
    }
}
```

```

        // milliseconds, and then repeats this process
        // indefinitely.
while (status != TERMINATE) {
    synchronized(this) {
        // As long as applet is stopped, don't do anything.
        while (status == SUSPEND)
            try { wait(); }
            catch (InterruptedException e) { }
    }
    switch ( (int)(4*Math.random()) ) {
        case 0: canvas0.randomColor(); break;
        case 1: canvas1.randomColor(); break;
        case 2: canvas2.randomColor(); break;
        case 3: canvas3.randomColor(); break;
    }
    synchronized(this) { // delay for a bit
        try { wait( (int)(3000*Math.random() + 30) ); }
        catch (InterruptedException e) { }
    }
} // end run()

} // end nested class Runner

public void init() {
    // Initialize the applet.  Create 4 ColorSwatchCanvasses
    // and arrange them in a horizontal grid.
    setBackground(Color.black);
    setLayout(new GridLayout(1,0,2,2));
    canvas0 = new ColorSwatchCanvas();
    canvas1 = new ColorSwatchCanvas();
    canvas2 = new ColorSwatchCanvas();
    canvas3 = new ColorSwatchCanvas();
    add(canvas0);
    add(canvas1);
    add(canvas2);
    add(canvas3);
}

public Insets getInsets() {
    // Put a 2-pixel black border around the applet.
    return new Insets(2,2,2,2);
}

public void start() {
    // Applet is being started or restarted.  Create a
    // thread or tell the existing thread to restart.
    status = GO;
    if (runner == null || ! runner.isAlive()) {
        runner = new Runner();
        runner.start();
    }
    else {
        runner.notify();
    }
}

```

```

    }

    public void stop() {
        // Applet is about to be stopped. Tell thread to suspend.
        synchronized(runner) {
            status = SUSPEND;
            runner.notify();
        }
    }

    public void destroy() {
        // Applet is about to be destroyed.
        // Tell thread to terminate.
        synchronized(runner) {
            status = TERMINATE;
            runner.notify();
        }
    }
} // end class RandomColorGridApplet

```

This is actually a cleaner way of programming threads than willy-nilly declaring all kinds of objects to be Runnable.

Note, by the way, how I did the synchronization in this example. Synchronization only works if all concerned threads are synchronized on the same object. In the `run()` method, synchronization is done on "this", which refers to the thread object. If I had simply declared `start()`, `stop()`, and `destroy()` to be synchronized methods, they would be synchronized on the applet object, not the thread object, which would do me no good at all. So instead, I use a `synchronized` statement to synchronize on the thread object, `runner`.

The use of the special variable, `this`, in the `run()` method raises another issue. When used in the `Runner` class, `this` refers to the object of type `Runner`. What about the `RandomColorGridApplet` object that is associated with the thread? Is there any way to refer to the applet object? Yes, but it's a little clumsy. Inside the nested `Runner` class, the applet object that is associated with the thread object, `this`, is referred to as `RandomColorGridApplet.this`.

Event-handling is an even more natural application of nested classes. Instead of adding the responsibility of listening for events onto an object that already has some other well-defined responsibility, you can create an object belonging to a nested class that has handling certain events as its one-and-only, clearly-defined responsibility. Since the event-handling object belongs to a nested class, it has access to any data and methods that it needs from the containing class.

Here is an outline of how you could use a nested class to handle action events from a button:

```

public class MyApplet extends Applet {

    class ButtonHandler implements ActionListener {
        public void actionPerformed(ActionEvent evt) {
            . . . // Handle the action event.
                // (This method has full access to all
                // the members of the MyApplet class.)
        }
    } // end nested class ButtonHandler
}

```

```

        public void init() {
            ButtonHandler handler = new ButtonHandler();
            Button btn = new Button("Do It!");
            btn.addActionListener(handler);
            . . . // other initialization
        }

        . . . // other members of MyApplet

    } // end class MyApplet

```

Some of the listener interfaces, such as `MouseListener` and `ComponentListener`, include a lot of methods. The rules for interfaces say that when a class implements an interface, it must include a definition for each method declared in the interface. For example, if you are only be interested in using the `mousePressed()` method of the `MouseListener` interface, you end up including empty definitions for `mouseClicked()`, `mouseReleased()`, `mouseEntered()`, and `mouseExited()`. If you use a specially-created nested class to handle events, there is a way to avoid this. As a convenience, the package `java.awt.event` includes several **adapter classes**, such as `MouseAdapter` and `ComponentAdapter`. The `MouseAdapter` class is a trivial class that implements the `MouseListener` interface by defining each of the methods in that interface to be empty. To make your own mouse listener classes, you can extend the `MouseAdapter` class and override just those methods that you are interested in. `ComponentAdapter` and other adapter classes work in the same way.

For example, if you want to respond to changes in the size of a component, you could use a component listener based on `ComponentAdapter` to listen for `componentResized` events:

```

public class MyApplet extends Applet {

    class Resizer extends ComponentAdapter {
        public void componentResized(ComponentEvent evt) {
            . . . // Respond to new component size.
            // (This method has full access to all
            // the members of MyApplet.)
        }
        // No need to define the other ComponentListener Methods
    } // end nested class Resizer

    public void init() {
        MyCanvas canvas = new MyCanvas(); // Some component in whose
                                           // size we are interested.
        canvas.addComponentListener( new Resizer() );
        . . . // more initialization
    }

    . . . // more members of MyApplet

} end class MyApplet

```

In some cases, you might find yourself writing a nested class and then using that class in just a single line of your program. For example, the `Resizer` class in the above example might well be used only in the line `"canvas.addComponentListener(new Resizer());"`. 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. The intention of this expression is to refer to: "a new object of a class that is the same as **superclass-or-interface** but with these **methods-and-variables** added." An expression of this form can be used in any statement where a regular "new" could be used. For example:

```
canvas.addComponentListener(
    new ComponentAdapter() {
        public void componentResized(ComponentEvent evt) {
            . . . // Respond to new component size.
        }
    } // end of anonymous class definition
); // end of canvas.addComponentListener statement
```

This defines an anonymous class that is a subclass of `ComponentAdapter`, containing the given `componentResized()` method. It also creates an object belonging to that anonymous class. That object is assigned to be the listener for component events from the canvas. When the canvas changes size, the object's `componentResized()` method is called.

Note that it is possible to base an anonymous class on an interface, rather than a superclass. In this case, the anonymous class must implement the interface by defining all the methods that are declared in the interface.

For a final example, we'll return to the colored-square applet from earlier on this page. In addition to the spontaneous color changes in that applet, each of the colored squares will change color if you click on it. The squares are objects belonging to a class named `ColorSwatchCanvas`. This class sets up a mouse listener to listen for clicks on the canvas. It does this with an anonymous class:

```
class ColorSwatchCanvas extends Canvas {

    // A canvas that displays a random color.  It changes to
    // another random color if the user clicks on it, or if the
    // randomColor() method is called.

    ColorSwatchCanvas() { // constructor

        randomColor(); // Select the canvas's initial random color.

        addMouseListener( // Create an object to listen for mouse clicks.
            new MouseAdapter() {
                public void mousePressed(MouseEvent evt) {
                    // Change color when mouse is pressed.
                    randomColor();
                }
            }
        ); // end addMouseListener statement

    } // end constructor

    void randomColor() {
        // Change the color of the canvas to a random color.
        int r = (int)(256*Math.random());
        int g = (int)(256*Math.random());
        int b = (int)(256*Math.random());
    }
}
```

```
        setBackground(new Color(r,g,b));  
        repaint();  
    }  
  
} // end class ColorSwatchCanvas
```

Using anonymous nested classes is generally considered to be the best programming style for event handling.

[[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, a program can open an independent window by creating an object of type `Frame`. A `Frame` has a title, displayed in the title bar at the top of the window. It can have a **menu bar** containing one or more **pull-down menus**. A `Frame` is a `Container`, which means that it can hold other GUI components. The default layout manager for a `Frame` is a `BorderLayout`. A common way to design a frame is to add a single GUI component, such as a `Panel` or `Canvas`, in the layout's `Center` position so that it will fill the entire frame.

It is possible for an applet to create 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 displays just one small button on the page. When you click the "Launch ShapeDraw" button, a window will be opened. This is another version of the `ShapeDraw` program that you've seen several times already. This version has a pull-down menu named "Add Shape", which contains commands that you can use to add shapes to the window. (Note for Macintosh users: The menus for the frame might just be added to the list of menus at the top of the screen. Look there for the "Add Shape" menu if you don't see it at the top of the window.) Once a shape is in the window, you can drag it around. The color that will be used for newly created shapes is controlled by another pull-down menu named "Color". There is also a pop-up menu that appears when you click on a shape in the appropriate, platform-dependent way.

Sorry, but your browser
doesn't support Java.

The window in this example belongs to a class named `ShapeDrawFrame`, which is defined as a subclass of `Frame`. The applet creates the window with the statement

```
shapeDraw = new ShapeDrawFrame( );
```

The constructor sets up the structure of the window and makes the window appear on the screen. Once the window has been created, it runs independently of the applet and of any other windows. Of course, there can be some communication through shared variables, method calls, and events. In the sample applet above, for example, the name of the button changes from "Launch ShapeDraw" to "Close ShapeDraw" while the window is open. If the user clicks on the "Close ShapeDraw" button, the applet responds by closing the window. This is done by calling an appropriate method in the window object. On the other hand, if the user closes the window by clicking on its close box, an event is generated belonging to the class `WindowEvent`. The applet listens for such events so that it can change the name of the button back to "Launch ShapeDraw" when the window is closed. (You might be interested to see how all this is handled in the source code for the applet. You can find it in the file [ShapeDrawLauncher.java](http://math.hws.edu/javanotes/c7/s7.html).)

Fortunately, you don't have to learn a lot of new material to use frames. There are a few specialized methods in the `Frame` class that you should know about. You'll want to know how to use pull-down menus and menu bars. And there are a few important events that are generated by windows. Aside from that, programming with frames is the same as programming with applets. Many of the new features show up in the constructor for the `ShapeDrawFrame` class, so let's take a look at that:

```

public ShapeDrawFrame() {
    // Constructor. Create and open the window. The window
    // has a menu bar containing two menus, "Add Shape" and
    // "Color". The content of the window is a canvas belonging
    // to the class ShapeCanvas.

    setBackground(Color.white);

    /* Set window title by calling the superclass constructor. */

    super("Shape Draw");

    /* Create a canvas that fills the frame. */

    canvas = new ShapeCanvas();
    add(canvas, BorderLayout.CENTER);

    /* Create pull-down menus and a menu bar. The frame listens
       for ActionEvents from the menus. These are generated when
       the user selects commands from the menus. */

    Menu colorMenu = new Menu("Color", true); // Color choice menu.
    colorMenu.add("Red");
    colorMenu.add("Green");
    colorMenu.add("Blue");
    colorMenu.add("Cyan");
    colorMenu.add("Magenta");
    colorMenu.add("Yellow");
    colorMenu.add("Black");
    colorMenu.add("White");

    colorMenu.addActionListener(this);

    Menu shapeMenu = new Menu("Add Shape", true); // Shape menu.
    shapeMenu.add("Rectangle");
    shapeMenu.add("Oval");
    shapeMenu.add("Round Rect");

    shapeMenu.addActionListener(this);

    MenuBar mbar = new MenuBar(); // Create menu bar and add the menus.
    mbar.add(shapeMenu);
    mbar.add(colorMenu);

    setMenuBar(mbar); // Add the menu bar to the frame.

    /* Do the remaining setup and show the window. */

    setBounds(30,50,380,280); // Set size and position of window.

    setResizable(false); // Make the window non-resizable.

    addWindowListener(
        new WindowAdapter() {
            // A listener that will close the window
            // when the user clicks its close box.

```

```

        public void windowClosing(WindowEvent evt) {
            ShapeDrawFrame.this.dispose();
        }
    }; // end addWindowListener statement

    show(); // Make the window visible.

} // end constructor

```

This constructor begins by calling a constructor from the superclass, `Frame`. The `Frame` constructor specifies a title for the window, which is displayed in the window's title bar. A `Frame` object contains an instance method, `setTitle(String)`, that can be used to set the title of the frame at any time. There is also a `getTitle()` method that returns the current title.

The frame's "Color" menu is created with the command `Menu colorMenu = new Menu("Color", true);`. The second parameter of the `Menu` constructor specifies that this should be a **tear-off menu**, which can be dragged by the user away from the menu bar, where it becomes a little independent window. (This might not work on all platforms.) This second parameter is optional. The `add()` method is used to add commands to the menu. The version of the `add()` command that is used here takes a `String`, which will appear as one of the commands in the menu. You can be more sophisticated than this if you want. There is another version of the `add` command that adds one menu as a **submenu** or **hierarchical menu** in another menu. You can add a separator line to a menu with the `addSeparator()` method. You can also create an object belonging to the class `MenuItem` and add it to a menu. A `MenuItem` represents a command, and the text of that command appears in the menu. But you can also associate a keyboard shortcut key with the menu item (like Control-V for pasting in Windows or Command-V for pasting on a Macintosh). Furthermore, a `MenuItem` can be enabled and disabled. And there are even `CheckboxMenuItems` representing menu items that can be checked and unchecked by the user (generally to represent options that can be turned on and off). You can look up all the details in a Java reference if you are interested.

For a menu to be useful, you must register an `ActionListener` with it. Whenever the user selects an item from the menu, an `ActionEvent` is generated and the `actionPerformed()` method of the action listener is called. The action command that is associated with the `ActionEvent` is the text of the menu item. (It is also possible to listen for action events from individual `MenuItem` objects, but it is more common simply to use one listener for the menu as a whole.) In the above constructor, the frame itself is registered as the listener for the menus. There is an `actionPerformed()` method elsewhere in the `ShapeDrawFrame` class that handles menu commands.

A pull-down menu can only appear in the menu bar of a frame. A menu bar is an object of type `MenuBar`. Menus are added to the menu bar using the `add()` method of the `MenuBar` object. The menu bar itself must be associated with the frame by calling the frame's `setMenuBar()` command. Note that a frame can have only one menu bar, but that you can change to a different menu bar at any time by calling `setMenuBar()`.

In this example, I call `setBounds(30, 50, 380, 280)` to set the position and size of the frame. The upper left corner of the frame will be at the point (30,50) on the computer screen, where (0,0) is the upper left corner of the screen. The width of the frame will be 380 and the height will be 280. By calling `setResizable(false)`, I make it impossible for the user to change the size of the frame. By default, the frame would be resizable. In most cases, it's better to allow the user to change the size of the frame, but that means you have to write a program that can deal with components that change size.

(Using the `setBounds()` command is also not particularly good style. Better style would be to use the `pack()` command, which tells the frame to set itself to its so-called **preferred size**. A frame computes its preferred size by taking into consideration the preferred sizes and the layout of all the components that it contains. However, I failed to define a `getPreferredSize()` method in the `ShapeCanvas` class, so

the canvas component in the frame doesn't know how big it wants to be. In this case, `setBounds()` works well enough. But if you ever start writing "serious" GUI components that will see a lot of reuse by yourself or other programmers, you have to start worrying about the fine points of style, like `getPreferredSize()`. The `MessageDialog` example that is discussed at the end of this page uses both `pack()` and `getPreferredSize()`. Look at the source code if you'd like to see how they work.)

The next statement in the constructor sets up a listener to listen for `WindowEvents` from the frame:

```
addWindowListener(
    new WindowAdapter() {
        // A listener that will close the window
        // when the user clicks its close box
        public void windowClosing(WindowEvent evt) {
            ShapeDrawFrame.this.dispose();
        }
    }
); // end addWindowListener statement
```

This requires some explanation. To understand it all, you need to have absorbed the material about anonymous nested classes and adapter classes from the [previous section](#). Frames generate events belonging to the class `WindowEvent`. There are seven types of `WindowEvent`, and the associated `WindowListener` interface declares seven methods. Here, I only want to define a response to one type of window event, so I base my listener object on the `WindowAdapter` class. The listener object belongs to an anonymous subclass of `WindowAdapter`. The `windowClosing()` method is called when the user clicks on the close box of the frame. The system does **not automatically close the window when the user does this! It just generates a `windowClosing` event. The event handler for that event is responsible for closing the window. (It can even decide not to close it. For example, the event handler method might ask the user, "Do you really want to close this window?" and give the user a chance to change his mind.) A frame is closed by calling its `dispose()` method. The statement `ShapeDrawFrame.this.dispose()` in the event handler refers to the `dispose()` method in the `ShapeDrawFrame` class.**

The final line in the constructor for `ShapeDrawFrame` is `"show();"`. This is extremely important. When a window is first created, it is hidden. The `show()` command makes the window appear on the screen.

By the way, the `ShapeDrawFrame` class also includes the following `main()` routine:

```
public static void main(String[] args) {
    new ShapeDrawFrame();
}
```

This means that `ShapeDrawFrame` can be executed as an independent program. When it is executed, the `main()` routine simply opens a window of type `ShapeDrawFrame`. After opening the window, the `main()` routine ends, but the window stays open until the user closes it. (If the frame is being run as an independent program, it would be a good idea to call `System.exit(0)` when the user closes the window. This statement will end the program completely, not just dispose of the window. This can be done in the `windowClosing` method.)

A problem sometimes arises when using `Insets` with frames. To leave a border between the edges of the frame and the components that it contains, you have to define a `getInsets()` method in the frame class. However, on some platforms, a frame already uses insets to leave space for the menu bar. Your `getInsets()` has to allow for this space. The way to do this is to call `super.getInsets()` to find out how much space the frame is already allowing for insets. Then add on the extra space that you want to allow for the border. The following `getInsets()` method adds an extra 2 pixels to the insets along each edge of the frame:

```

public Insets getInsets() {
    Insets currentInsets = super.getInsets();
    Insets myInsets = new Insets();
    myInsets.top = currentInsets.top + 2;
    myInsets.bottom = currentInsets.bottom + 2;
    myInsets.left = currentInsets.left + 2;
    myInsets.right = currentInsets.right + 2;
    return myInsets;
}

```

(When you call `super.getInsets()`, you are not supposed to modify the object that it returns. You should construct a new object and return that one, as is done in this example.)

Dialogs

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, 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.

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 or even bring the parent to the front of the screen 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 Java, a dialog box is an object belonging to the class `Dialog` or to one of its subclasses. A `Dialog` cannot have a menu bar, but it can contain other GUI components, and it generates the same `WindowEvents` as a `Frame`. `Dialog` objects can be either modal or modeless. A parameter to the constructor specifies which it should be.

Click on this button to run a little demonstration of modal dialogs:

Sorry, but your browser
doesn't support Java.

The dialogs in this example belong to a class named `MessageDialog`, which is defined in the file [MessageDialog.java](#). You might find this class useful for creating simple dialogs for your own programs. The source code for the applet and frame classes used in this example is in the file [DialogDemoLauncher.java](#)

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

Section 7.8

Looking Forward: Swing and Java 2

JAVA PLATFORM 2.0 HAS BEEN OUT for over a year. Java Platform 2.0 is the name that Sun Microsystems uses, somewhat confusingly, for version 1.2 of the Java language plus some optional features that make it suitable for large-scale business applications. In the transition from Java 1.0 to Java 1.1, many features were "deprecated," meaning that it is no longer advisable to use them. The same was not true in the transition from Java 1.1 to 1.2. With only a very few minor exceptions, everything in Java 1.1 carries over unchanged to Java 1.2. This textbook covers Java 1.1, but everything in it is valid for Java 1.2 and Java Platform 2.

Even in Java 1.1, there are many features that I will not cover. These features take the form of standard classes that provide various capabilities. For example, Java 1.1 includes a set of classes that are used for RMI (**Remote Method Invocation**). RMI allows a program running on one computer on a network to use objects that exist on other computers on the network. The methods of that object are "invoked", that is called, remotely.

One especially important advanced feature of Java 1.1 is the **Java Bean** technology. A Java Bean is simply an object that follows certain programming guidelines. All the GUI components in `java.awt` are Beans, for example. An object that follows the guidelines can be used in a Bean Builder application, which is a visual development tool that lets application developers assemble programs from pre-existing parts. The parts are Java Beans. Java Beans have **properties**. A property is just a value associated with the bean. A property has a name. If `PropName` is a property, then the bean contains one or both of the methods `getPropName()` and `setPropName()` for reading and changing the value of the property. To define a property named `PropName`, the bean doesn't have to do anything more than define these methods. This is one of the programming guidelines for beans. A Bean Builder application can recognize a property by looking for these methods, and it will make it possible for an application developer to manipulate these properties.

To make it easier to work with properties, Java 1.1 defines a `PropertyChangeEvent` class and an associated `PropertyChangeListener`. A bean can generate a `PropertyChangeEvent` when one of its properties changes. It can define an `addPropertyChangeListener()` method so that other objects can be registered to receive notification when the value of a property changes. A Bean Builder can check for this method to determine whether a bean supports property change events.

Perhaps the most visible change from Java 1.1 to Java 1.2 is the introduction of a new set of GUI components called "Swing." Swing components can be used as an alternative to the components defined in the AWT. Swing uses the idea of properties and `PropertyChangeEvents` extensively. It also adds a similar event type, `ChangeEvent`, which carries less information than a `PropertyChangeEvent` and can therefore be handled more efficiently.

In addition to Swing, Java 1.2 introduces a few other new features. For example, it includes classes to support Drag and Drop (the ability to drag information from one component and drop it on another, even between Java and non-Java programs) and Accessibility (classes that support access to Java programs, including Swing components, by tools that such as audible text readers, which can help to make a program accessible to the blind).

Java 1.2 also includes a greatly enhanced two-dimensional graphics capability. (A three-dimensional graphics package is available as an optional add-on.) This takes the form of a new class, `Graphics2D`, which is a subclass of the `Graphics` class that is used in the AWT. In addition to the graphics routines from the `Graphics` class, a `Graphics2D` object includes support for additional shapes, curves, lines that are more than one pixel thick, and dotted and dashed lines. In addition, it has support for geometric transformations, which means that it is possible to apply rotation, scaling, and translation operations to

whatever is drawn in the graphics context.

Swing

Swing consists of a collection of classes defined in the package `javax.swing`. Swing builds on the basic functionality of the AWT, including the `Component` and `Container` classes and all the layout manager, event, and listener classes, but it replaces the actual GUI components from the AWT with a new set of GUI components. For each type of GUI component in the AWT, there is a corresponding class in Swing. The `java.awt.Button` class is replaced by `javax.swing.JButton`. The `java.awt.Frame` class is replaced by `javax.swing.JFrame`. The `java.applet.Applet` class is replaced by `javax.swing.JApplet`. And so on. There are a few differences between the AWT components and the corresponding Swing components, so it is not quite possible to take an existing AWT program and convert it into a Swing program by adding a "J" to all the class names. But the other changes that are needed are minor. One significant change is that it is not possible to add components directly to a `JApplet` or `JFrame`. The components have to be added to something called the "content pane" with a command such as `getContentPane().add(comp);`. Another difference is that there is no `JCanvas` class. A Swing program would most likely use a `JPanel` as a replacement for an AWT canvas.

However, Swing extends these AWT-like components with a large number of new component classes. For example, there is a `JTable` class for making two-dimensional tables like those used in a spreadsheet. There is a `JTabbedPane` that is something like a `CardLayout` except that there are "tabs" on the tops of the cards that the user can click to move from one card to another. A `JToolBar` consists of a row or column of icons that the user can click to perform various commands. A `JSplitPane` is divided into two sections, with a divider between them. The user can move the divider to change the sizes of the two sections. There is a text editor pane -- something like a `TextArea` -- that supports a mixture of different sizes and styles of text. You've probably seen all these things in modern GUI programs, but they are not available in the AWT.

Furthermore, the components in Swing have more capabilities individually than those in the AWT. For example, the `JPanel` class can support double buffering automatically. Buttons and labels can display images instead of or in addition to text. A component can have an associated "tool tip" that pops up in a little window when the user points the mouse at the component.

So, you might be asking, why not just use Swing instead of the AWT exclusively? It might be that in the future, everyone will do so. However, there are still some reasons to use the AWT. For one thing, the enhanced capabilities of Swing come at a cost of increased complexity. Programming for any GUI is hard. Programming for a modern, full-featured GUI like that implemented by Swing is proportionately harder. Furthermore, Java 1.2 is still not as widely available in Web browsers as Java 1.1, and it is such a large system that it is not clear how quickly it will be deployed. Java 1.1 seems to me to be quite suitable for the kind of small application that is likely to appear on a Web page. It is also my guess that it will take some time for a system as large and complex as Swing to become completely stable.

Another question that arises is, why was Swing implemented as a separate package? Why not just add more components to the AWT? The reason here is that there are a lot of architectural (under-the-hood) innovations in Swing. A Swing component is really a different sort of thing from an AWT component, even though they have similar looks and behaviors. It is not possible to mix Swing and AWT components in the same program (unless you really, really know why you are doing). Every AWT component has a so-called "native peer", which is the GUI component that you actually see on the screen. This peer is created and controlled by the native operating system (Windows, Mac OS, Unix). In an AWT program, there are many such native components. In a Swing, there is only one, corresponding to the entire window or applet. The other Swing components are written entirely in Java. They are drawn using Java graphics. Their events are processed entirely by methods written in Java. This means that Swing components should work more consistently on different platforms.

Furthermore, a Swing component has a double structure. It consists of a **model** and a **user interface**

delegate, or UI delegate. The model is an object that holds the component's data, such as the text in a `JTextArea` or the numbers in a `JTable`. The UI delegate is an object that is in charge of drawing the component and handling events associated with that component. It is possible to provide different views of the same data by using the same model for several components. The model can communicate with a UI delegate with `ChangeEvent`s. When the data in the model changes, a `ChangeEvent` informs the UI delegate, and it redraws the component to reflect the change. This structure makes it possible for Swing to implement "pluggable look-and-feel". It's possible to change the whole visual style of a program by installing a new set of UI delegates, without changing the underlying data in the models. This makes it possible for a program to look and feel like a Windows program on a Windows computer, while the same program looks and feels like a Macintosh program on a Mac. It is even possible to change the look-and-feel of a program while it is running.

Obviously, this has been only the briefest of introductions to Swing. Actually using it requires ascending a steep learning curve. However, I think you'd find that what you've learned about the AWT is a good foundation for learning about Swing.

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 use 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. 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 scroll bars. 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 `Panel` instead of `Applet`. 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 canvas 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'll need instance variables to refer to the buttons. One problem you have to deal with is that the buttons are used in both the applet class and the canvas class.

I strongly advise using a subroutine 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 `Button`, 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 `TextField` to the strip of controls along the bottom of the applet. The user can enter the bet in this `TextField`. When the game begins, check the amount of the bet. You should do this when the game begins, not when it ends, because several errors can occur: The contents of the `TextField` might not be a legal number. The bet that the user places might be more money than the user has, or 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 `TextField` uneditable while the game is in progress. If `betInput` is the `TextField`, you can make it editable and uneditable by the user with the commands `betAmount.setEditable(true)` and `betAmount.setEditable(false)`.

In the `paint()` 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. In the constructor for the canvas class, you should not call `doNewGame()`. You might want to start the game with the message "Welcome to Blackjack".

[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 `Thread`. Add a `Thread` to the original source code, [StopWatch.java](#), to display the elapsed time in seconds. Create the thread in the `mousePressed()` routine when the timer is started. Stop the thread in the `mousePressed()` routine when the timer 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. In my `run()` method, I insert a delay of 100 milliseconds after I set the text. Here is an applet that tests my solution to this exercise:

[See the solution!](#)

Exercise 7.7: The applet at the end of [Section 7.8](#) 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 `Choice` 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 `putFigure()` subroutine that draws all the figures. You can add a `putMultiFigure()` routine to draw a figure and some or all of its reflections. `putMultiFigure` can call the existing `putFigure` to draw each figure. It decides which reflections to draw based on the setting of the symmetry `Choice` menu. Where the `mousePressed`, `mouseDragged`, and `mouseReleased` methods call `putFigure`, they should call `putMultiFigure` instead.

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 Choice 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 in a `Frame`. This is not an easy exercise, since the material on frames in [Section 7.7](#) is sort of sketchy. The information is there if you read carefully. (But I won't think too badly of you if you just look at the solution.)

As another improvement, you can add an "Undo" button. 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, `OSC`, is created, make a second off-screen canvas, `undoBuffer`, of the same size. Before starting any drawing operation, copy the image from `OSC` to `undoBuffer`. You can do this with the commands

```
Graphics undoGr = undoBuffer.getGraphics();
undoGr.drawImage(OSC, 0, 0, null);
```

When the user clicks "Undo", just swap the values of `OSC` and `undoBuffer` and repaint. The previous image will appear on the screen. Clicking on "Undo" again will "undo the undo".

Here is a button that opens the paint program in its own window. (You don't have to write an applet like this one. Just open 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 canvas* can be used to do *double buffering*. Explain this. (What are off-screen canvases? How are they used? Why are they important? What does this have to do with animation?)

Question 3: One of the main classes in the AWT is the `Component` 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 `Checkbox` and how is it used?

Question 7: What is a *thread*?

Question 8: If you want to program an applet to show an animation, you have to create a thread. Why?

Question 9: What is meant by a *non-static nested class*? What are the advantages of using one?

Question 10: What is the purpose of the `Frame` 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 type of data structure (although it is in fact more than this, since it also includes operations or methods for working with that data). However, this type of data structure -- consisting of a fairly small number of named instance variables -- is only one of the many different types of data structure that a programmer might need. In many cases, the programmer has to build more complicated data structures by linking objects together. But there is one type of data structure that is so important and so basic that it is built into every programming language: the array.

An **array** is a data structure consisting of a numbered list of items, where all the items are of the same type. In Java, the items in an array are always numbered from zero up to some maximum value, which is set when the array is created. For example, an array might contain 100 integers, numbered from zero to 99. The items in an array can belong to one of Java's primitive types. They can also be references to objects, so that you could, for example, make an array containing all the `Buttons` in an applet.

This chapter discusses how arrays are created and used in Java. It also covers the standard class `java.util.Vector`. An object of type `Vector` is very similar to an array object.

Contents of Chapter 8:

- Section 1: [Creating and Using Arrays](#)
 - Section 2: [Programming with Arrays](#)
 - Section 3: [Vectors and Dynamic Arrays](#)
 - 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,

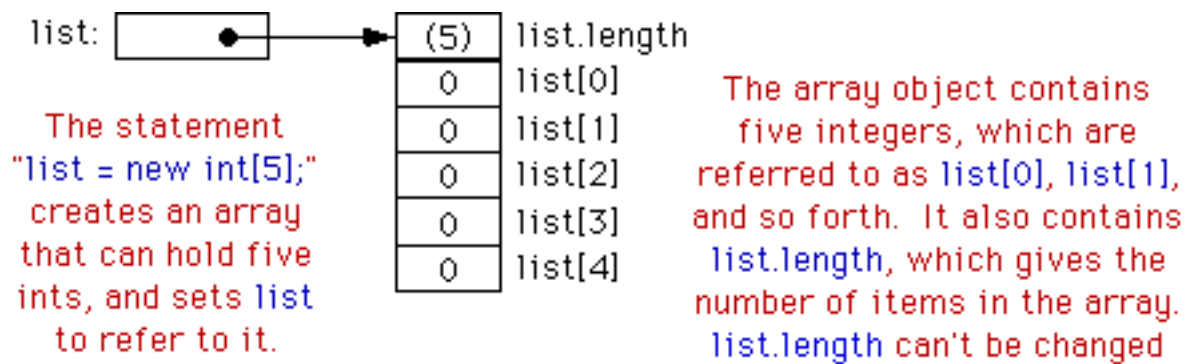
```
int[] list;
```

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 7.6](#).) 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 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 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 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?

Sorry, but your browser
doesn't support Java.

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.

Sorry, but your browser
doesn't support Java.

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 `paint()` 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 `paint()` method

```

public void paint(Graphics g) {
    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.
}
```

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.) 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 `paint()` method for the applet becomes

```
public void paint(Graphics g) {
    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. Simple array indexing does the translation for us!

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

Section 8.3

Vectors and Dynamic Arrays

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](#).)

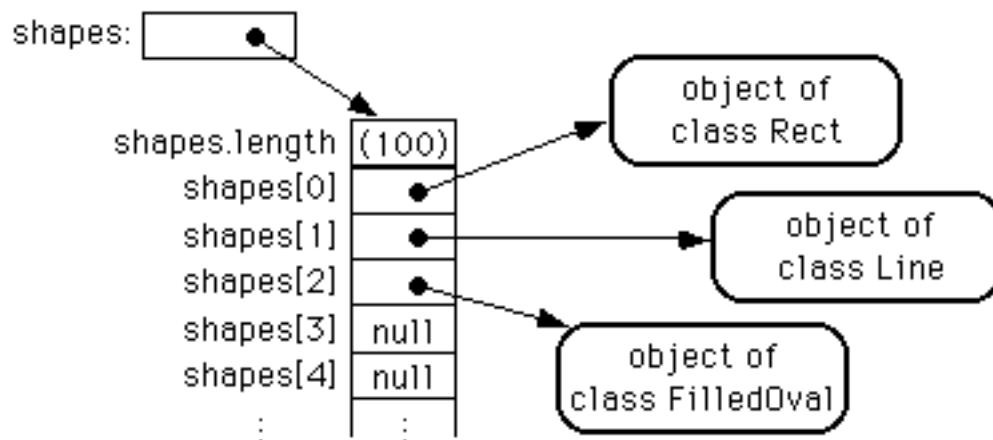
Then an array of type `Shape[]` can hold references to objects belonging to the subclasses of `Shape`. For example, the situation created by the statements

```

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.)

Sorry, but your browser
doesn't support Java.

Vectors

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. The closest we can come in Java is to use the 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 `Vector` that serves much the same purpose. The `Vector` class is in the package `java.util`, so if you want to use the `Vector` class in a program, you should put the directive `"import java.util.*;"` or `"import java.util.Vector;"` at the beginning of your source code file.

The `Vector` class differs from my `DynamicArrayOfInt` class in that a `Vector` object always has a definite size, and it is illegal to refer to a position in the `Vector` that lies outside its size. In this, a `Vector` is more like a regular array. However, the size of a `Vector` can be increased at will, and it increases automatically in certain cases. The `Vector` class defines many instance methods. I'll describe some of the most useful. Suppose that `vec` is a variable of type `Vector`.

`vec.size()` -- This function returns the current size of the vector. Valid positions in the vector are between 0 and `vec.size() - 1`. Note that the size can be zero, and it is zero when a vector is first created.

`vec.addElement(obj)` -- Adds an object onto the end of the vector, increasing the size of the vector by 1. The parameter, `obj`, can refer to an object of any type.

`vec.elementAt(N)` -- This function returns the value stored at position `N` in the vector. `N` must be an integer in the range 0 to `vec.size() - 1`. If `N` is outside this range, an error occurs.

`vec.setElementAt(obj, N)` -- Assigns the object, `obj`, to position `N` in the vector, replacing the item previously stored at position `N`. The integer `N` must be in the range from 0 to `vec.size() - 1`.

`vec.insertElementAt(obj, N)` -- Moves all the items in the vector in positions `N` and above up one position, and then assigns the object, `obj`, to position `N`. The integer `N` must be in the range from 0 to `vec.size()`. The size of the vector increases by 1.

`vec.removeElement(obj)` -- If the specified object occurs somewhere in the vector, it is removed from the vector. Any items in the vector that come after the removed item are moved down one position. The size of the vector decreases by 1. The value of the parameter, `obj`, must not be `null`.

`vec.removeElementAt(N)` -- Deletes the element at position `N` in the vector. `N` must be in the range 0 to `vec.size() - 1`. Items in the vector that come after position `N` are moved down one position. The size of the vector decreases by 1.

`vec.setSize(N)` -- Changes the size of the vector to `N`, where `N` must be an integer greater than or equal to zero. If `N` is bigger than the current size of the vector, new elements are added to the vector and filled with `nulls`. If `N` is smaller than the current size, then the extra elements are removed from the end of the vector.

`vec.indexOf(obj, start)` -- A function that searches for the object, `obj`, in the vector, starting at position `start`. If the object is found in the vector at or after position `start`, then the position number where it is found is returned. If the object is not found, then -1 is returned. The second parameter can be omitted, and the search will start at position 0.

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 a `Vector` named `players`. This variable would be declared as

```
Vector players;
```

and initialized to refer to a new, empty `Vector` object with

```
players = new Vector();
```

If `newPlayer` is a variable that refers to a `Player` object, the new player would be added to the game by saying

```
players.addElement(newPlayer);
```

and if player number *i* leaves the game, it is only necessary to say

```
players.removeElementAt(i);
```

All this works very nicely. The only slight difficulty arises when you use the function `players.elementAt(i)` to get the value stored at position *i* in the vector. 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 typecast it to type `Player`:

```
Player plr = (Player)players.elementAt(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.elementAt(i);
    plr.makeMove();
}
```

In [Section 5.4](#), I displayed an applet, `ShapeDraw`, that uses vectors. Here is another version of the same idea, simplified to make it easier to see how vectors are being used. Right-click the large white canvas to add a colored rectangle. (On a Macintosh, Command-click the canvas.) 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.

Sorry, but your browser
doesn't support Java.

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](#). Here, I just want to look at the parts of the program that use a vector.

The applet uses a variable named `rects`, of type `Vector`, to hold information about the rectangles that have been added to the drawing area. The objects that are stored in the vector 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 in the vector `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.elementAt(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 Vector 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.elementAt(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 canvas is simply `rects.removeElement(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 vector, the rectangle that is in the last position in the vector is in front of all the other rectangles on the screen. So we need to move the rectangle to the last position in the vector. This is done by removing the rectangle from its current position in the vector 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 Vector.
    if (rect != null) {
        rects.removeElement(rect);
        rects.addElement(rect);
        repaint();
    }
}

```

This should be enough to give you the basic idea. You can look in the source code for more details.

As these examples show, Vectors can be very useful. The package `java.util` also includes a few other classes for working with Objects. We'll look at some of them in later chapters.

[[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 of course you can say 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.

    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 to
        // indicate this.

        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 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.
    // A is assumed to be sorted into increasing order.

    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 a `Vector` 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) {
    // Assume that A contains itemsInArray items in increasing
    // order (A[0] <= A[1] <= ... <= A[itemsInArray-1]).
    // This routine adds newItem to the array in its proper
    // position, keeping the array in increasing order.

    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.

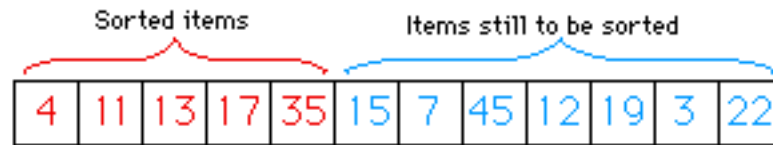
        int temp = A[itemsSorted]; // The item to be inserted.
        int loc = itemsSorted - 1; // Start at end of list.

        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.
        }

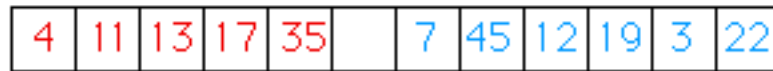
        A[loc + 1] = temp; // Put temp in last vacated space.
    }
}
```

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.

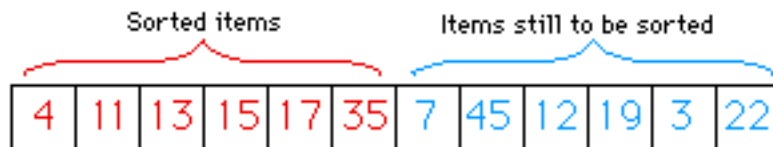
Start with a partially sorted list of items:



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`. 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) {
    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;
    }
}
```

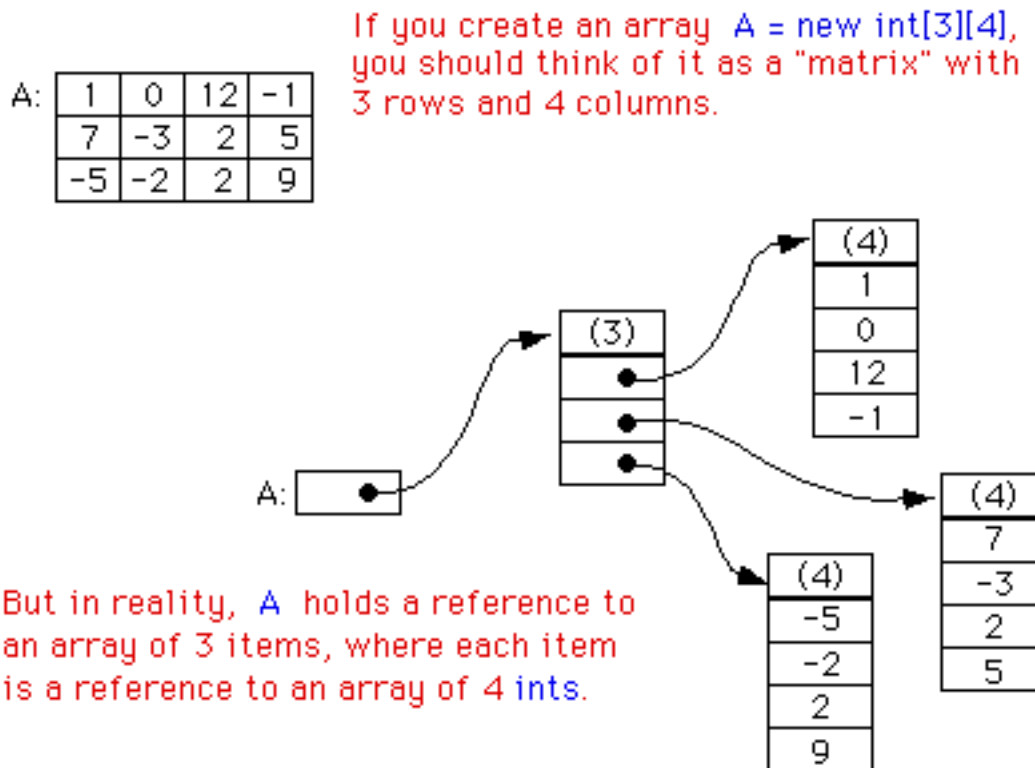
[[Next Section](#) | [Previous Section](#) | [Chapter Index](#) | [Main Index](#)]

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; j++)
        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 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 a 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 `paint()` method from the `MosaicCanvas` class, so you can see how it uses the array:

```
public void paint(Graphics g) {
    // Draw all the colored rectangles in the grid.
    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.

Sorry, but your browser
doesn't support Java.

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 every legal moves that is 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 `Vector` instead of an array to hold the moves as we find them. As we add moves to the vector, 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 vector
    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 two previous methods to check
    // whether the player can legally jump from (r1,c1) to (r3,c3).
    // It is assumed that the player has a piece at (r1,c1), that
    // (r3,c3) is a position that is 2 rows and 2 columns distant
    // from (r1,c1) and that (r2,c2) is the square between (r1,c1)
    // and (r3,c3).
}

```

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, vectors, 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;

    Vector moves = new Vector(); // Moves will be stored in this vector.

    /* 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 vector.
    */
}

```

```

    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.addElement(new CheckersMove(row, col, row+2, col+2));
                if (canJump(player, row, col, row-1, col+1, row-2, col+2))
                    moves.addElement(new CheckersMove(row, col, row-2, col+2));
                if (canJump(player, row, col, row+1, col-1, row+2, col-2))
                    moves.addElement(new CheckersMove(row, col, row+2, col-2));
                if (canJump(player, row, col, row-1, col-1, row-2, col-2))
                    moves.addElement(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 vector.
    */

    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.addElement(new CheckersMove(row, col, row+1, col+1));
                    if (canMove(player, row, col, row-1, col+1))
                        moves.addElement(new CheckersMove(row, col, row-1, col+1));
                    if (canMove(player, row, col, row+1, col-1))
                        moves.addElement(new CheckersMove(row, col, row+1, col-1));
                    if (canMove(player, row, col, row-1, col-1))
                        moves.addElement(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 vector 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.elementAt(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 `paint()` 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 nothing is drawn on the applet until you click the second vertex of the polygon. You have to 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, getSize().width, 0, getSize().height)"`. 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 [SimpleAnimationApplet](#), which was first introduced in [Section 3.7](#). Use an array of type `MovingBall[]` to hold the 25 balls. The `drawFrame()` method in your applet should move all the balls and draw them.

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: To do this exercise, you need to know the material on components and layouts from [Chapter 7](#). Write an applet that draws pie charts based on data entered by the user. A pie chart is a circle divided into colored wedges. Each wedge of the pie corresponds to one number in an array of positive numbers. The number of degrees in the wedge is proportional to the corresponding number. If `g` is a variable of type `Graphics`, then a wedge can be drawn with the command

```
g.fillArc(left, top, width, height, startAngle, degrees);
```

All the parameters in this command are integers. The first four parameters give the left edge, top edge, width, and height of a rectangle containing the "pie". For a circle, the width and height must be the same. The fifth parameter tells the starting angle of the wedge, and the sixth tells the number of degrees in the wedge.

Your applet should include 12 `TextFields` where the user can enter the data for the pie chart. You will need an array of type `TextField[]` to keep track of these input boxes. [Section 7.4](#) has an example that shows how to get a number of type `double` from an input box. You should ignore any input boxes that are empty. Use the data from the non-empty input boxes for the pie chart.

The applet should have a button that the user clicks to draw the pie chart. And it should have a sub-class of the `Canvas` class that will display the pie chart. I suggest that this class include an instance variable of type `int[]` to store the data needed for the pie chart. The data that you need is the angles of the dividing lines between the wedges. This is not the same as the data from the input boxes, but it can be computed from that data. Suppose the user's data is `data[0], data[1], ..., data[dataCt-1]`. Let `dataSum` be the sum of all the user's data, `data[0] + data[1] + ... + data[dataCt-1]`. Let `sum` be the sum of the first `i` data values, `data[0] + data[1] + ... + data[i-1]`. Then the angle for the `i`-th dividing line in the pie chart is given by `"(int)(360*sum/dataSum + 0.5)"`. (The "360" is there because it represents the number of degrees in a full circle. The "+0.5" is there so the answer will be rounded to the nearest integer rather than truncated downwards.)

Here is my solution:

[See the solution!](#)

Exercise 8.6: 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 hilite the legal moves. You will only need two classes, a short applet class to set up the applet and a `GoMokuCanvas` 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!](#)

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

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.

Recently, as I write this, the failure of two multi-million space missions to Mars has been 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.

For example, consider the 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);
```

Preconditions and postconditions are, by the way, a convenient way to express the contract of a subroutine. The preconditions of a subroutine are the conditions that must be satisfied for the subroutine to work correctly. The postcondition expresses the task that is accomplished by the subroutine. Here, for example, is a subroutine that searches for a number in an array. The contract is stated in the comment in terms of a precondition and a postcondition.

```
static int find( double[] A, double x ) {

    // Search for x in the array A.
    // Precondition:   A is not null.
    // Postcondition:  The return value is -1 if x does not
    //                  occur in the array. Otherwise, the
    //                  return value is an index in A where
    //                  x is found.

    int i; // An index in the array A.

    i = 0;
    while ( i < A.length && A[i] != x ) {
```

```

        i++;
    }

    if (i == A.length)
        return -1;
    else
        return i;
} // end find()

```

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 the 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') {
        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:

Sorry, but your browser
doesn't support Java.

[[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

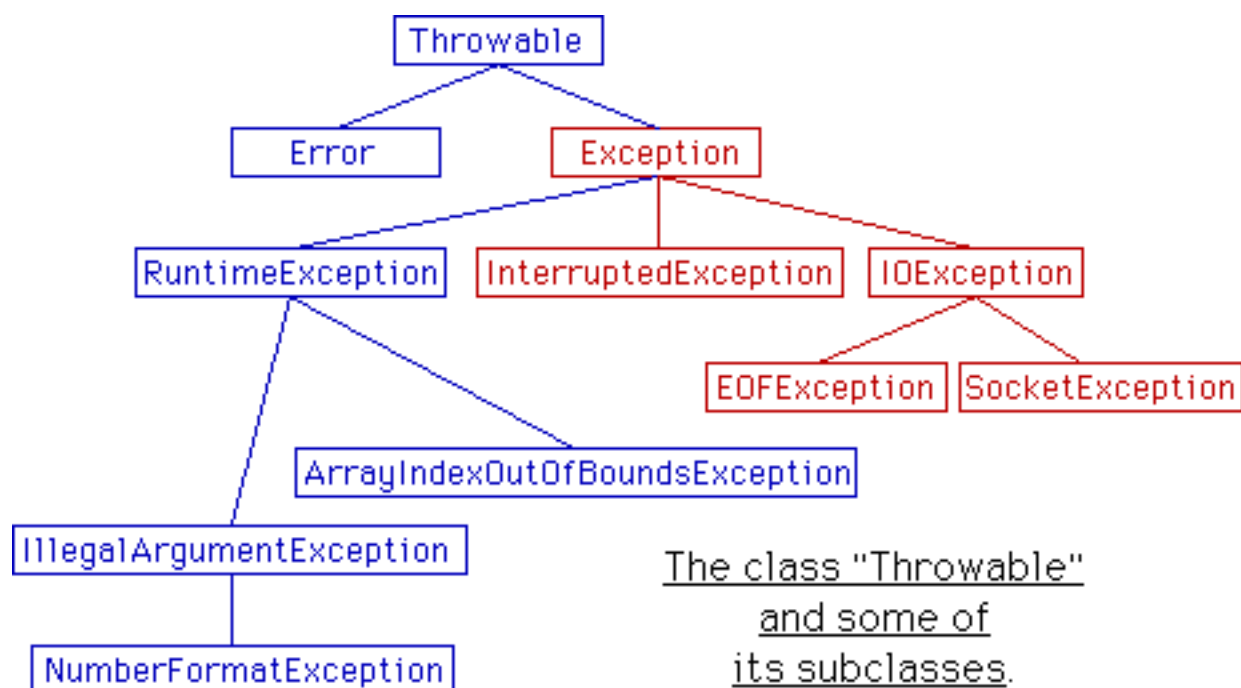
subclasses. In general, each different type of exception is represented by its own subclass of `Throwable`. `Throwable` has two direct subclasses, `Error` and `Exception`. These two subclasses in turn have many other predefined subclasses. In addition, a programmer can create new exception classes to 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 to 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 `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. One possibility is to place the statement in a `try` statement that has a `catch` clause that handles the exception. The other possibility 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 `BigInteger`s are objects, they must be manipulated using instance methods from the `BigInteger` class. For example, you can't add two `BigInteger`s with the `+` operator. Instead, if `N` and `M` are variables that refer to `BigInteger`s, 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 `TextField` generates an `ActionEvent` when the user presses return.)

The applet will need a canvas for displaying the graph. To keep things simple, the canvas should represent a fixed region in the xy -plane, defined by $-5 \leq x \leq 5$ 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. Some of the examples in this chapter will also use advanced techniques from [Chapter 7](#), such as threads, Frames, and nested classes.

Contents of Chapter 10:

- 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](#)
-

[[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 7.6](#).) `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 `RuntimeException`s. 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 exception-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 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 File Dialogs

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 Macintosh computer.
- `examples/data.dat` -- a relative path name under UNIX. "Examples" is the name of a directory that is contained within the current directory, and data.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.

In many cases, however, you would like 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. Java provides a platform-independent method for using file dialog boxes in the form of a class called `FileDialog`. This class is part of the package `java.awt`.

There are really two types of file dialog windows: one for selecting an existing file to be used for input, and one for specifying a file for output. You can specify which type of file dialog you want in the constructor

for the `FileDialog` object, which has the form

```
public FileDialog(Frame parent, String title, int mode)
```

where `parent` is presumably the main window of the program, `title` is meant to be a short string describing the dialog box, and `mode` is one of the constants `FileDialog.SAVE` or `FileDialog.LOAD`. Use `FileDialog.SAVE` if you want an output file, and use `FileDialog.LOAD` if you want a file for input. You can actually omit the `mode` parameter, which is equivalent to using `FileDialog.LOAD`.

Once you have a `FileDialog` object, you can use its `show()` method to make it appear on the screen. It will stay on the screen until the user either selects a file or cancels the request. The instance method `getFile()` can then be called to retrieve the name of the file selected by the user. If the user has canceled the file dialog, then the `String` value returned by `getFile()` will be `null`. Since the user can select a file that is not in the current directory, you will also need directory information, which can be retrieved by calling the method `getDirectory()`. For example, if you want the user to select a file for output, and if the main window for your application is `mainWin`, you could say:

```
FileDialog fd = new FileDialog(mainWin, "Select output file",
                               FileDialog.SAVE);

fd.show();
String fileName = fd.getFile();
String directory = fd.getDirectory();

if (fileName != null) { // (otherwise, the user canceled the request)

    ... // open the file, save the data, then close the file

}
```

Once you have the file name and directory information, you will have to combine them into a usable file specification. The best way to do this is to create an object of type `File`. The `File` object can then be used as a parameter in a constructor for a `FileReader`, `FileWriter`, `FileInputStream`, or `FileOutputStream`. For example, the body of the `if` statement in the above example could include:

```
try {
    File file = new File(directory, fileName);
    PrintWriter out = new PrintWriter(new FileWriter(file));
    ... // write the data to the output stream, out
    out.close();
}
catch { IOException e }
    ... // respond to the exception
}
```

The `File` class has other uses as well. An object of type `File` really represents a file name rather than a file. The file to which the name refers might or might not exist. If it does exist, it might actually be a directory rather than a regular file. A `File` object can be constructed from a directory and a file name, as in the above example. The directory can be specified as a `String` or as an object of type `File` that refers to a directory. A `File` object can also be created just from a file name, as in the constructor call `new File("data.txt")`. In that case, the directory is assumed to be the current directory.

`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 this 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
```

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;
}
```

```
/* 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. */

    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 `TextArea` 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. Menus and windows for standalone programs were discussed in [Section 7.7](#). The program also uses file dialogs, which were introduced in [Section 2](#).

When the user selects the Save command from the File menu, the program pops up a file dialog box where the user specifies the file. The text from the `TextArea` 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.
    FileDialog fd;          // A file dialog that lets the user
                           // specify the file.
    String fileName;        // Name of file specified by the user.
    String directory;       // Directory that contains the file.
    fd = new FileDialog(this, "Save Text to File", FileDialog.SAVE);
    fd.show();
    fileName = fd.getFile();
    if (fileName == null) {
        // The fileName is null if the user has canceled the file
        // dialog. In this case, there is nothing to do, so quit.
        return;
    }
    directory = fd.getDirectory();
    try {
        // Create a PrintWriter for writing to the specified
        // file and write the text from the window to that stream.
        File file = new File(directory, fileName);
        PrintWriter out = new PrintWriter(new FileWriter(file));
        String contents = text.getText();
        out.print(contents);
        out.close();
    }
    catch (IOException e) {
        // Some error has occurred while opening or closing the file.
        // Show an error message.
        new MessageDialog(this, "Error:  " + e.toString());
    }
}
```

The `MessageDialog` class that is used at the end of this method pops up a dialog box to display the error message to the user. In a GUI program, it wouldn't make much sense to write the error to standard output, since the user is not likely to be paying attention to standard output, even if it is visible on the screen. `MessageDialog` is not a standard part of Java. It is defined in the file [MessageDialog.java](#).

(By the way, there is one problem with this method, with illustrates the difficulties of dealing with text files on different computing platforms. The lines in a `TextArea` are separated by line feed characters, `\n`. This is the standard for separating lines in the UNIX operating system, but both Macintosh and Windows have different standards. Macintosh uses the carriage return character, `\r`, as a line separator, while Windows uses the two-character sequence `\r\n`. The `doSave()` method writes the line feed characters from the `TextArea` to the output file. On Macintosh and Windows computers, the output file will not be in the proper format for a text file -- assuming that the implementation of `PrintWriter` in your version of Java

doesn't take care of the problem. The `TextReader` class, which is used in the method that opens files, can handle any of the three formats of text files.)

When the user selects the Open command, a 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 `TextAreas` 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 `TextArea` is removed. Then lines are read from the file and appended to the `TextArea` 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 TextArea.
    FileDialog fd;        // A file dialog that lets the user
                        // specify the file.
    String fileName;      // Name of file specified by the user.
    String directory;     // Directory that contains the file.
    fd = new FileDialog(this, "Load File", FileDialog.LOAD);
    fd.show();
    fileName = fd.getFile();
    if (fileName == null) {
        // The fileName is null if the user has canceled the file
        // dialog. In this case, there is nothing to do, so quit.
        return;
    }
    directory = fd.getDirectory();
    try {
        // Read lines from the file until end-of-file is detected,
        // or until 100 lines have been read. The lines are appended
        // to the TextArea, with a line feed after each line. The
        // test for end-of-file in the while loop is
        // in.peek() != '\0' because calling in.eof() could skip
        // over and discard any blank lines in the file. Blank lines
        // should be copied to the TextArea just like any other lines.
        File file = new File(directory, fileName);
        TextReader in = new TextReader(new FileReader(file));
        String line;
        text.setText("");
        int lineCt = 0;
        while (lineCt < 100 && in.peek() != '\0') {
            line = in.getln();
            text.appendText(line + '\n');
            lineCt++;
        }
        if (in.eof() == false) {
            text.appendText(
                "\n\n***** Text truncated to 100 lines! *****\n");
        }
        in.close();
    }
    catch (Exception e) {
        // Some error has occurred while opening or closing the file.
        // Show an error message.
    }
}
```



```
        new MessageDialog(this, "Error:  " + e.toString());  
    }  
}
```

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. 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 class for this style of networking is called `URL`. 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.

The second style of I/O is much more low-level, and 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. Two-way communication over a network involves two sockets, one on each of the computers involved in the communication. Java uses a class called `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 the `URL` and `Socket` classes, and shows how they relate to input and output streams and to exceptions. The networking classes discussed in this section, including `URL` and `Socket`, are defined in the package `java.net`.

The URL Class

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. If you have a `URL` object, you can use it to open a network connection to the resource at that address. The `URL` class, and an associated class called `URLConnection`, provide a large number of methods for working with such connections. One of the methods in the `URL` class, `getContent()`, allows you retrieve the resource that the `URL` points to with a single command. That is, if `url` is an object of type `URL`, then `url.getContent()` is an `Object` that contains the text file, image, or other resource found on the Web at the given url. For example, if the resource is a standard Web page in HTML format, then the `Object` returned by `getContent()` might be a `String` that contains the actual HTML code that describes the page. Alternatively, since the type of object that is returned can depend on the implementation of Java that you are using, the object might be an `InputStream` or a `Reader` from which you can read the contents of the HTML page.

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. So of course you should call the constructor inside a `try` statement and handle the potential `MalformedURLException` in a `catch` clause.

When you write an applet, there are two methods 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 address = 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 use the `getContent()` method to retrieve the data stored at that url. Alternatively, you can use the method `openStream()` from the `URL` class to obtain an `InputStream` from which you can read the data using any available input method. For example, if `address` is an object of type `URL`, you could simply say

```
InputStream in = address.openStream();
```

to get the input stream. This method does all the work of opening a network connection. When you read from the input stream, it does all the hard work of obtaining data over that connection. Ordinarily, you would wrap the `InputStream` object in a `DataInputStream` or `TextReader` and do all your input through that.

Various exceptions can be thrown as the attempt is made to open the connection and read data from it. Most of these exception are of type `IOException`, and such errors must be caught and handled. But these operations can also cause **security exceptions**. An object of type `SecurityException` is thrown when a program attempts to perform some operation that it does not have permission to perform. For example, a Web browser is typically configured to forbid an applet from making a network connection to any computer other than the computer from which the applet was downloaded. If an applet attempts to connect to some other computer, a `SecurityException` is thrown. A security exception can be caught and handled like any other exception. It does not require mandatory exception handling.

To put this all together, let's consider an applet that reads data from a url over the network. The url's `getContent()` method is used to retrieve the data. If the data is of type `String`, then the string is displayed in a `TextArea`. If `getContent()` returns an input stream, then the data is read from that stream and displayed. If `getContent()` returns any other type of object, then just the name of the class to which the object belongs is displayed. The data is read by a separate thread. (Threads were discussed in [Section 7.5](#).) Here is the `run()` method that is executed by the thread to do the work. A few things here will be unfamiliar to you, but you should be able to figure them out.

```

public void run() {
    // Loads the data in the url specified by an instance variable
    // named urlName. The data is displayed in a TextArea named
    // textDisplay. Exception handling is used to detect and respond
    // to errors that might occur.

    try {

        /* Create the URL. This can throw a MalformedURLException. */

        URL url = new URL(getDocumentBase(), urlName);

        /* Get the object that represents the content of the URL. This
           can throw an IOException or a SecurityException. */

        Object content = url.getContent();

        /* Display the content in the TextArea. If it is not of type
           String, InputStream, or Reader, just display the name of the
           class to which the content belongs. For an InputStream
           or Reader, at most 10000 characters are read. For efficiency,
           characters are read from a stream using a BufferedReader. */

        if (content instanceof String) {
            // Show the text in the TextArea.
            textDisplay.setText((String)content);
        }
        else if (content instanceof InputStream
                  || content instanceof Reader) {
            // Read up to 10000 characters from the stream, and
            // show them in the TextArea.
            textDisplay.setText("Receiving data...");
            BufferedReader in; // for reading from the URL stream
            if (content instanceof InputStream) {
                in = new BufferedReader(
                    new InputStreamReader( (InputStream)content ) );
            }
            else if (content instanceof BufferedReader) {
                in = (BufferedReader)content;
            }
            else {
                in = new BufferedReader( (Reader)content );
            }
            char[] data = new char[10000]; // The characters read.
            int index = 0; // Number of chars read.
            while (true) {
                // Read up to 10000 chars and store them in the array.
                if (index == 10000)
                    break;
                int ch = in.read();
                if (ch == -1)
                    break;
                data[index] = (char)ch;
                index++;
            }
        }
    }
}

```

```

        if (index == 0)
            textDisplay.setText("Couldn't get any data!");
        else
            textDisplay.setText(new String(data,0,index));
        in.close();
    }
    else {
        // Set text area to show what type of data was read.
        textDisplay.setText("Loaded data of type\n    "
                            + content.getClass().getName());
    }
}
catch (MalformedURLException e) {
    // Can be thrown when URL is created.
    textDisplay.setText(
        "\nERROR!  Improper syntax given for the URL to be loaded.");
}
catch (SecurityException e) {
    // Can be thrown when the connection is created.
    textDisplay.setText("\nSECURITY ERROR!  Can't access that URL.");
}
catch (IOException e) {
    // Can be thrown while data is being read.
    textDisplay.setText(
        "\nINPUT ERROR! Problem reading data from that URL:\n"
        + e.toString());
}
finally {
    // This part is done, no matter what, before the thread ends.
    // Set up the user interface of the applet so the user can
    // enter another URL.
    loadButton.setEnabled(true);
    inputBox.setEditable(true);
    inputBox.selectAll();
    inputBox.requestFocus();
}
} // end of run() method

```

And here is the working applet that uses this routine. When you press the "Load" button, the applet will try to load data from the url specified in the text-input box. When the applet first starts up, it contains a url for its own source code. If you click the Load button, it will try to load and display that source code. (If there is a problem, and if you would still like to see the complete source code, you can find it in the file [URLExampleApplet.java](http://math.hws.edu/javanotes/c10/s4.html).)

Sorry, your browser
doesn't support Java.

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 generating other urls and seeing 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.)

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 used 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.

This client/server model, in which there is one server program that supports multiple clients, is a perfect application for threads. A server program has one main thread that manages the listening socket. This thread runs continuously as long as the server is in operation. Whenever the server socket receives a connection request from a client, the main thread makes a new thread to handle the communications with that particular client. This client thread will run only as long as the client stays connected. The server thread and any active client threads all run simultaneously, in parallel. A server that works in this way is said to be **multithreaded**. Client programs, on the other hand, tend to be simpler, having just one network connection and just one thread (although there is nothing to stop a program from using several client sockets at the same time, or even a mixture of client sockets and server sockets).

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 **port number** is used in addition to the Internet 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. (This is why a server needs a separate thread just to wait for connection requests.) 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. Each time the server receives a connection request, it should create a new thread to handle the connection with the client. Suppose that you've written a method `createServiceThread(Socket)` that creates such a thread. Then a simple version of the `run()` method for the main server thread would be:

```
public run() {
    try {
        ServerSocket server = new ServerSocket(1728);
        while (true) {
            Socket connection = server.accept();
            createServiceThread(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. (This means that even when you write a client program, you might want to use a separate thread to handle the connection, so that the program can continue to respond to user inputs while the connection is being established. Otherwise, the program will just freeze for some indefinite period of time.) Once the connection is established, you can use the 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.) We'll look at some examples in the [next section](#).

Section 10.5

Programming Networked Applications

SOCKETS AND CLIENT/SERVER programming were introduced in the [previous section](#) in a mostly theoretical way. This section presents a few complete programming examples. All but one of the examples in this section are standalone programs, so you won't see them running on this Web page. If you want to run these programs, you will have to compile the source code and run them with a Java interpreter.

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 DataClient math.hws.edu". (You might need to replace the "java" command with another command if you are using a different Java interpreter.) 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 port 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. In the previous section, I noted that a server typically opens a separate thread to handle each connection. However, in this simple example, the server just calls a subroutine. 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 sendData() 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();
            sendData(connection);
        }
    }
    catch (Exception e) {
        TextIO.putln("Sorry, the server has shut down.");
        TextIO.putln("Error:  " + e);
        return;
    }

} // end main()


static void sendData(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 prepended
                                     // to every 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 reading incoming 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 1, the rest of the line is a message from one user to the other. If the character is 0, 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.

There are several problems with the `CLChat` programs. 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. My next example is a GUI chat program. The class `ConnectionWindow`, which you will find in the source code file [ConnectionWindow.java](#), supports two-way chatting between two users over the Internet. The `ConnectionWindow` 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.

A `ConnectionWindow` has a text-input box where the user can type messages that are to be sent to the partner on other side of the network connection. The user can send a message at any time by pressing return in the text-input box. At the same time, the user on the other end of the connection can do the same thing. How are these messages received? Each `ConnectionWindow` runs a separate thread that reads the messages sent from the other side of the connection. (Both the sent and received messages are displayed to the user in a large `TextArea` that acts as a transcript of the conversation.) Thus the sending and receiving of messages are handled by different subroutines which are executed by different threads. The two processes do not interfere with each other. The code for sending and receiving individual messages is essentially the same as the code in the `CLChat` programs.

There remains the question of how a connection can be established between two `ConnectionWindows`. As the `ConnectionWindow` class is designed, the connection must be established before the window is opened. The connected `Socket` is passed as a parameter to the `ConnectionWindow` constructor. This makes `ConnectionWindow` into a nicely reusable class that can be used in a variety of programs. 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 `ConnectionWindow` 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 ConnectionWindow");
    }
    System.out.println("Connected.");
}
catch (Exception e) {
    System.out.println("Error opening connection.");
    System.out.println(e.toString());
    return;
}

ConnectionWindow w;    // The window for this end of the connection.
w = new ConnectionWindow("ConnectionWindow", connection);

```

As it happens, I've taken the rather twisty approach of putting this `main()` routine in the `ConnectionWindow` class! This means that you can run `ConnectionWindow` as a standalone program. If you run it with the command `"java ConnectionWindow -s"`, it will run as a server. To run it as a client, use the command `"java ConnectionWindow <server>"`, where `<server>` is the name or IP number of the computer where the server is running. I say that this approach is twisty because the `main()` routine could easily be in another class. It has nothing to do with the function of `ConnectionWindow` objects. (This is an illustration of the separation between the static and the non-static parts of a class -- a discussion that you might remember from earlier in this text.)

There is still a big problem with running `ConnectionWindow` in the way I've just described. Suppose I want to set up a connection. How do I know who has a `ConnectionWindow` server running? 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. Once the connection is open, the server reads a command from the client. It understands three types of commands:

- A REGISTER command that adds a client to the list of available chatters. The server keeps this list in an internal data structure. The connection remains open and the client waits for some other user to request a connection with that client. Once a connection is made, the client is removed from the list -- the server does not support multiple connections to the same chatter.
- 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, you will see an error message at the top of the applet. It might say something like "Connection refused" or "Network not reachable". If you have downloaded this on-line textbook and are reading the copy on your own computer, you should be able to run the ConnectionBroker server on your computer and use the applet connect to it. Here is the applet:

Sorry, but your browser
doesn't support Java.

If the applet does find a server, it will display the list of available chatters in the large area in the center of the applet. If no chatters are available on the server, then you'll just see the message "No connections available" at the top of the applet. If you want to establish a connection, click on one of the items in the list of chatters and then click the "Connect" button. A ConnectionWindow will open which you can use to chat with the selected chatter (unless some error occurs). As the applet runs, the list displayed in the applet might become out of date, so a person in the list might no longer be available. To get a new, updated list from the server, click the "Refresh" button.

If you want to add yourself to the list, enter your name or some other information about yourself in the text-input box at the bottom of the applet, and click on the button labeled "Register me with this info:". The text that you entered in the box appears in the list, and a window opens to await the incoming connection. This window contains the message "Waiting for connection". If some user of the BrokeredChat applet requests a connection with you, the window will display the message "Connected" and you can start chatting with that person. You can close the connection window at any time. If you do this before a connection request is received, this will remove you from the list of chatters on the server.

You can enter yourself multiple times in the list, if you want, and you can connect to multiple people on the list. A separate ConnectionWindow will open for each connection. You can even connect with yourself, so you can try out the applet even if you are the only user.

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 [ConnectionWindow.java](#).

By the way, the ConnectionBroker server does not require that the clients that connect to it be chat programs. Once a connection relay has been set up between two clients, the server will happily transmit any messages that are sent through it. The only rule that the server enforces is that if either client sends a

message that begins with the `CLOSE` command character, `]`, then the connection will be closed. So the `ConnectionBroker` could easily support, say, a game of checkers between two clients instead of a chat session.

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 at the end of [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 FINAL 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 structured 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](#) | [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()
```

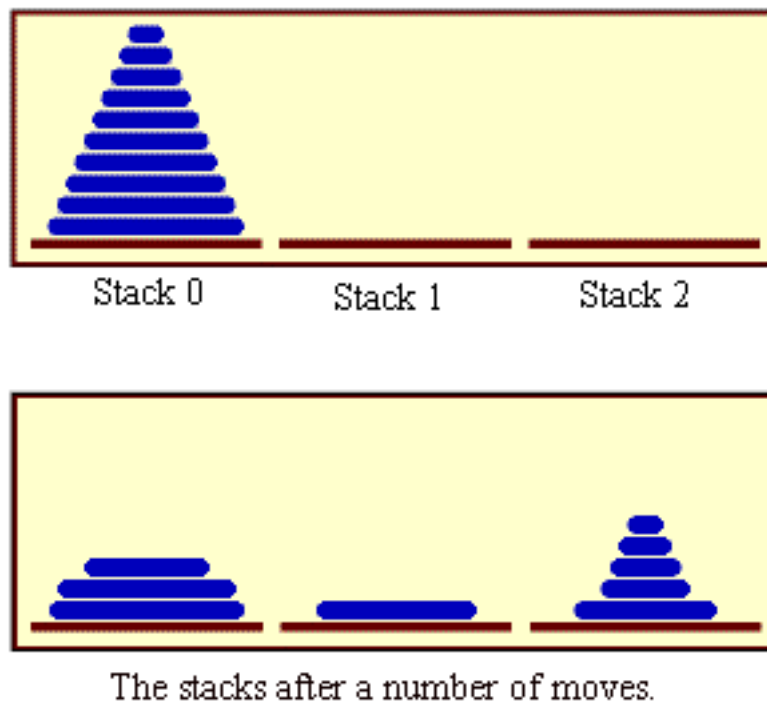
In this routine, the parameters `loIndex` and `hiIndex` specify the part of the array that is to be searched. To search an entire array, it is only necessary to call `binarySearch(A, 0, A.length - 1, value)`. In the two base cases -- where there are no elements in the specified range of indices and when 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 as a spare location. Can we reduce this to smaller problems of the same type, possibly generalizing the problem a bit to make this possible? It seems natural to consider the size of the problem to be the number of disks to be moved. If there are N disks in Stack 0, we know that we will eventually have to move the bottom disk from Stack 0 to Stack 1. But before we can do that, according to the rules, the first $N-1$ disks must be on Stack 2. Once we've moved the N -th disk to Stack 1, we must move the other $N-1$ disks from Stack 2 to Stack 1 to complete the solution. But moving $N-1$ disks is the same type of problem as moving N disks, except that it's a smaller version of the problem. This is exactly what we need to do recursion! The problem has to be generalized a bit, because the smaller problems involve moving disks from Stack 0 to Stack 2 or from Stack 2 to Stack 1, instead of from Stack 0 to Stack 1. In the recursive subroutine that solves the problem, the stacks that serve as the source and destination of the disks have to be specified. It's also

convenient to specify the stack that is to be used as a spare, even though we could figure that out from the other two parameters. The base case is when there is only one disk to be moved. The solution in this case is trivial: Just move the disk in one step. Here is a version of the subroutine that will print out step-by-step instructions for solving the problem:

```
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.

Sorry, but your browser
doesn't support Java.

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 simply, 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. 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 high, 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 apply the same procedure. As part of that procedure, you have to move down one row and apply the same procedure. 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 `paint()` 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." );
    }
}

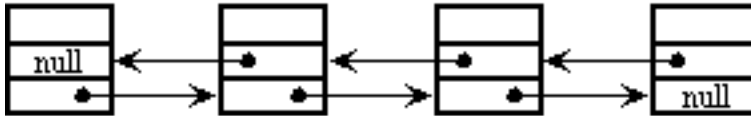
```

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.

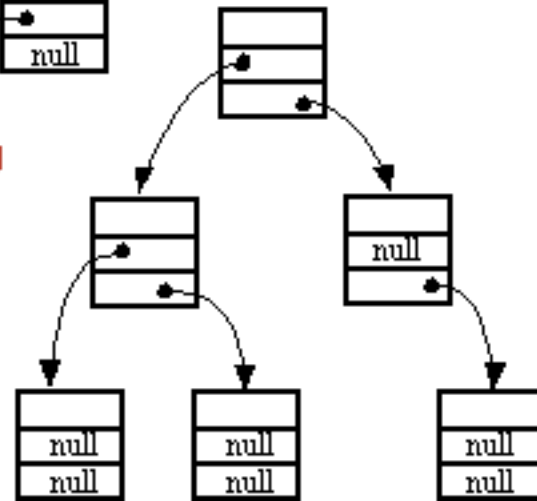
In this example, the supervisor variable is quite natural and useful. In fact, data structures that are built by linking objects together are so useful that they are a major topic of study in computer science. We'll be looking at a few typical examples. In this section and the [next](#), we'll be looking 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;
}
```

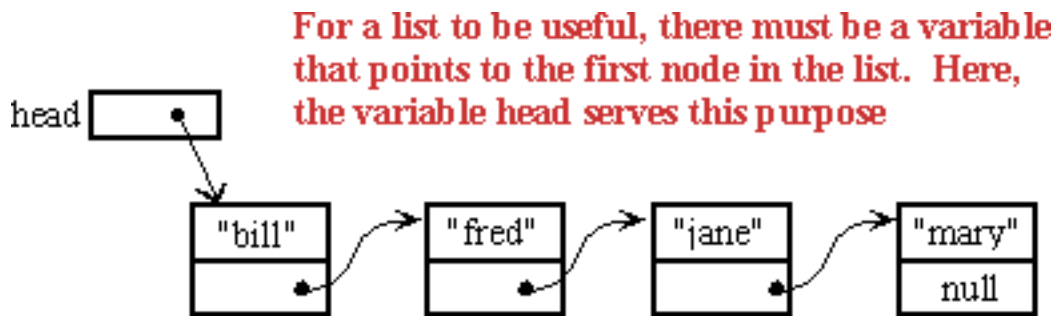
The term **node** is often used to refer to one of the objects in a linked data structure. Objects of type `Node` can be chained together as shown in the top part of the above picture. The last node in such a list can always be identified by the fact that the instance variable `next` in the last node 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.

Sorry, but your browser
doesn't support Java.

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

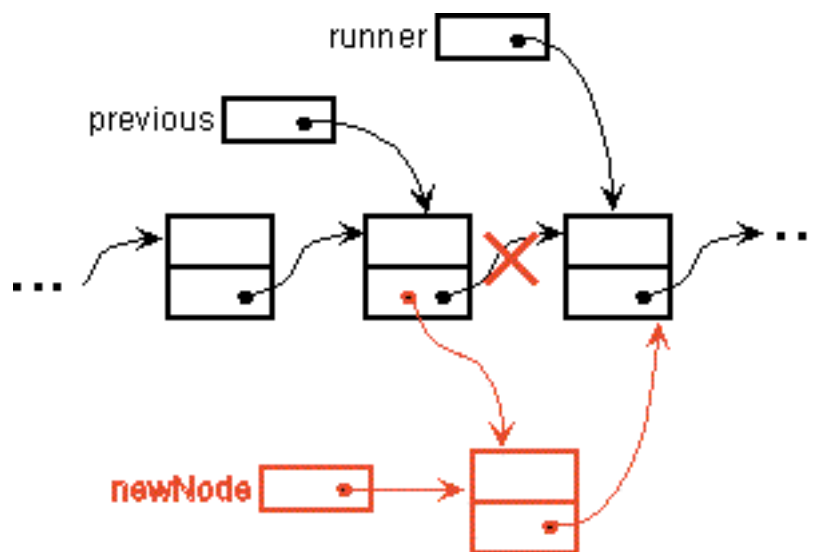
```

    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 `Vector`, 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.
```

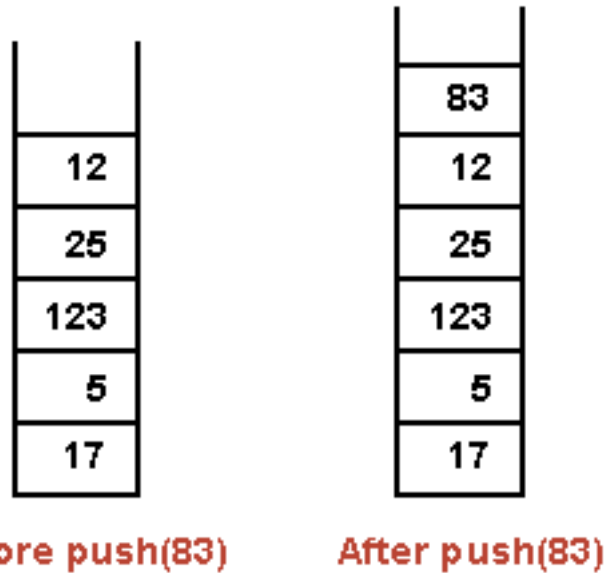
```
int pop()  -- Remove the top int from the stack and return it.
```

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 7.6](#) 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;
    }

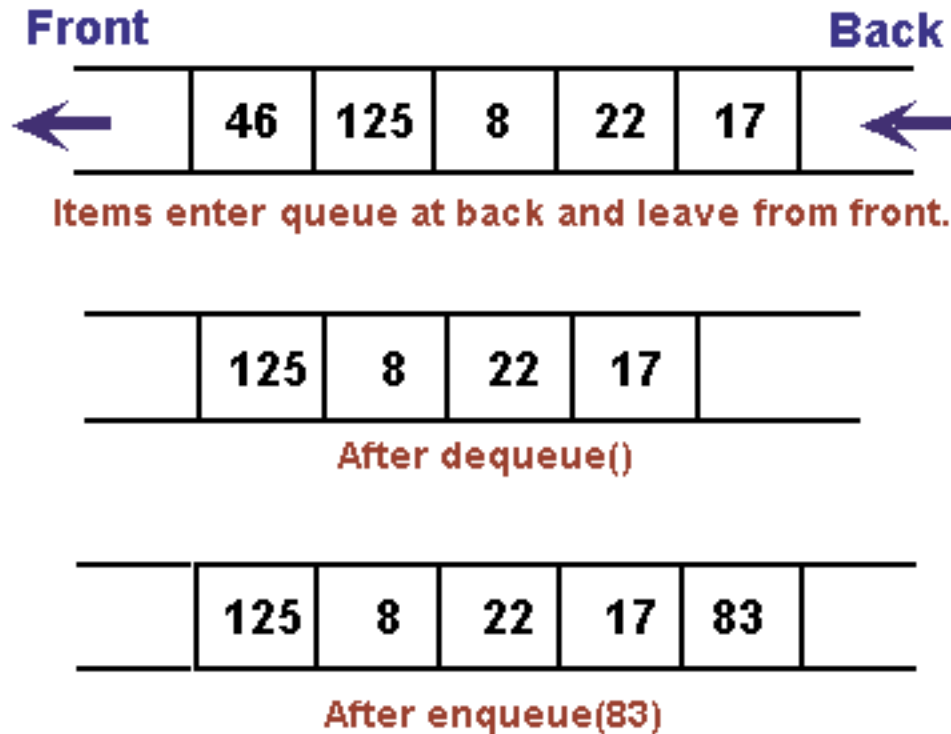
    public boolean isEmpty() {
        // Returns true if the stack is empty. Returns false
        // if there are one or more items on the stack.
        return (top == 0);
    }
} // end class StackOfInts

```

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](#).

Sorry, but your browser
doesn't support Java.

Queues seem very natural because they occur so often in real life, but there are times when stacks are appropriate and even essential. For example, consider what happens when a routine calls a subroutine. The first routine is suspended while the subroutine is executed, and it will continue only when the subroutine returns. Now, suppose that the subroutine calls a second subroutine, and the second subroutine calls a third, and so on. Each subroutine is suspended while the subsequent subroutines are executed. The computer has to keep track of all the subroutines that are suspended. It does this with a stack.

When a subroutine is called, an **activation record** is created for that subroutine. The activation record contains information relevant to the execution of the subroutine, such as its local variables 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:

Sorry, but your browser
doesn't support Java.

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:

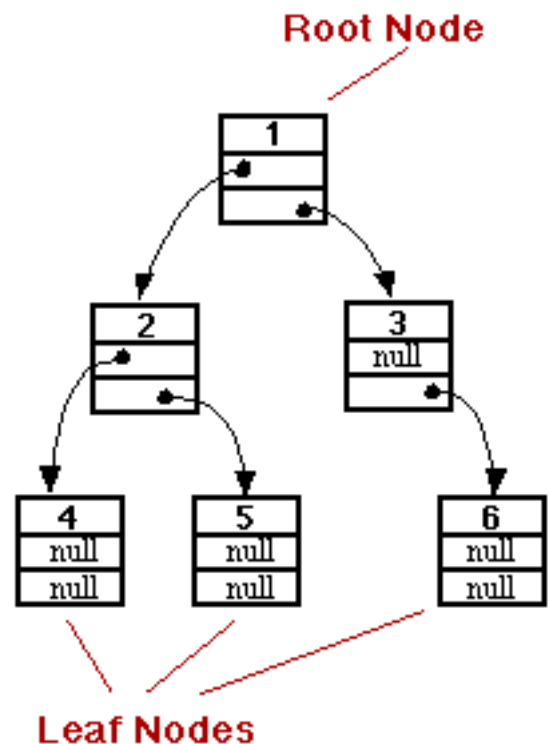
```
class TreeNode {
    int item;           // The data in this node.
    TreeNode left;      // Pointer to the left subtree.
    TreeNode right;     // Pointer to the right subtree.
}
```

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 descendents (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

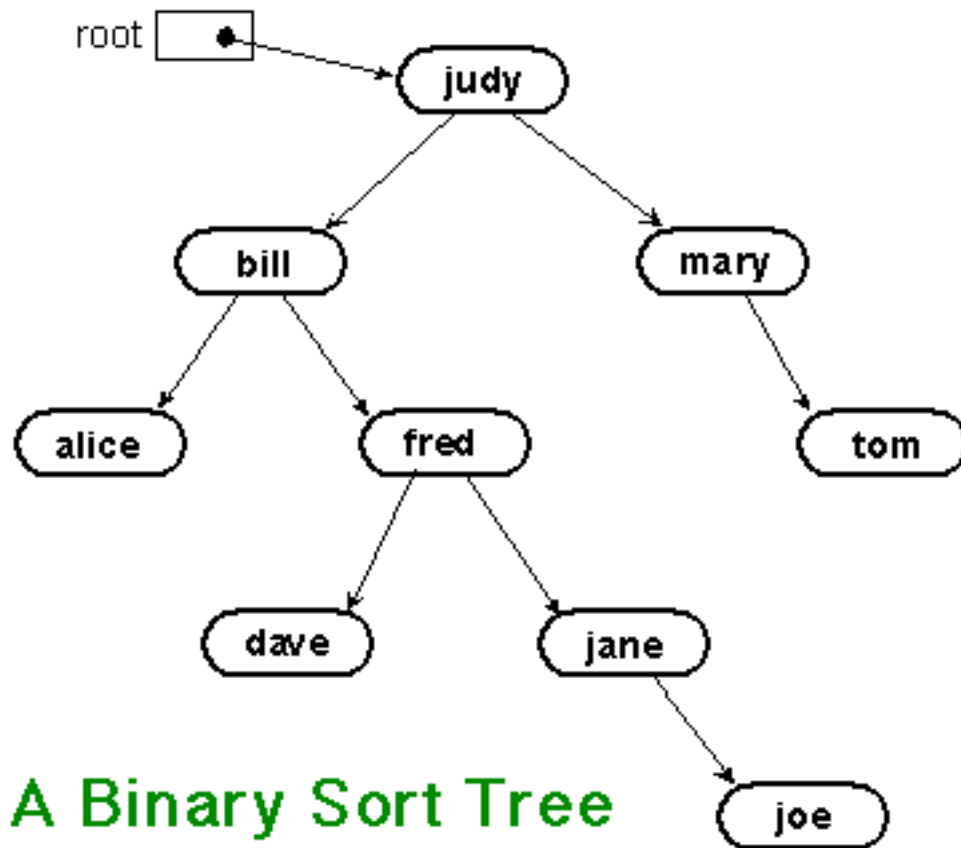
```

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:

Sorry, but your browser
doesn't support Java.

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(runner.item) ) {
            // We've found the item.
            return true;
        }
        else if ( item.compareTo(runner.item) < 0 ) {
            // If the item occurs, it must be in the left subtree,
            // So, advance the runner down one level to the left.
            runner = runner.left;
        }
        else {
            // If the item occurs, it must be in the right subtree.
            // So, advance the runner down one level to the right.
            runner = runner.right;
        }
    } // end while
} // end treeContainsNR();
```

The subroutine for inserting a new item into the tree turns out to be more similar to the non-recursive search routine than to the recursive. The insertion routine has to handle the case where the tree is empty. In that case, the value of `root` must be changed to point to a node that contains the new item:

```
root = new TreeNode( newItem );
```

But this means, effectively, that the `root` can't be passed as a parameter to the subroutine, because it is impossible for a subroutine to change the value stored in an actual parameter. (I should note that this is something that **is possible in other languages.**) **Recursion uses parameters in an essential way. There are 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**.

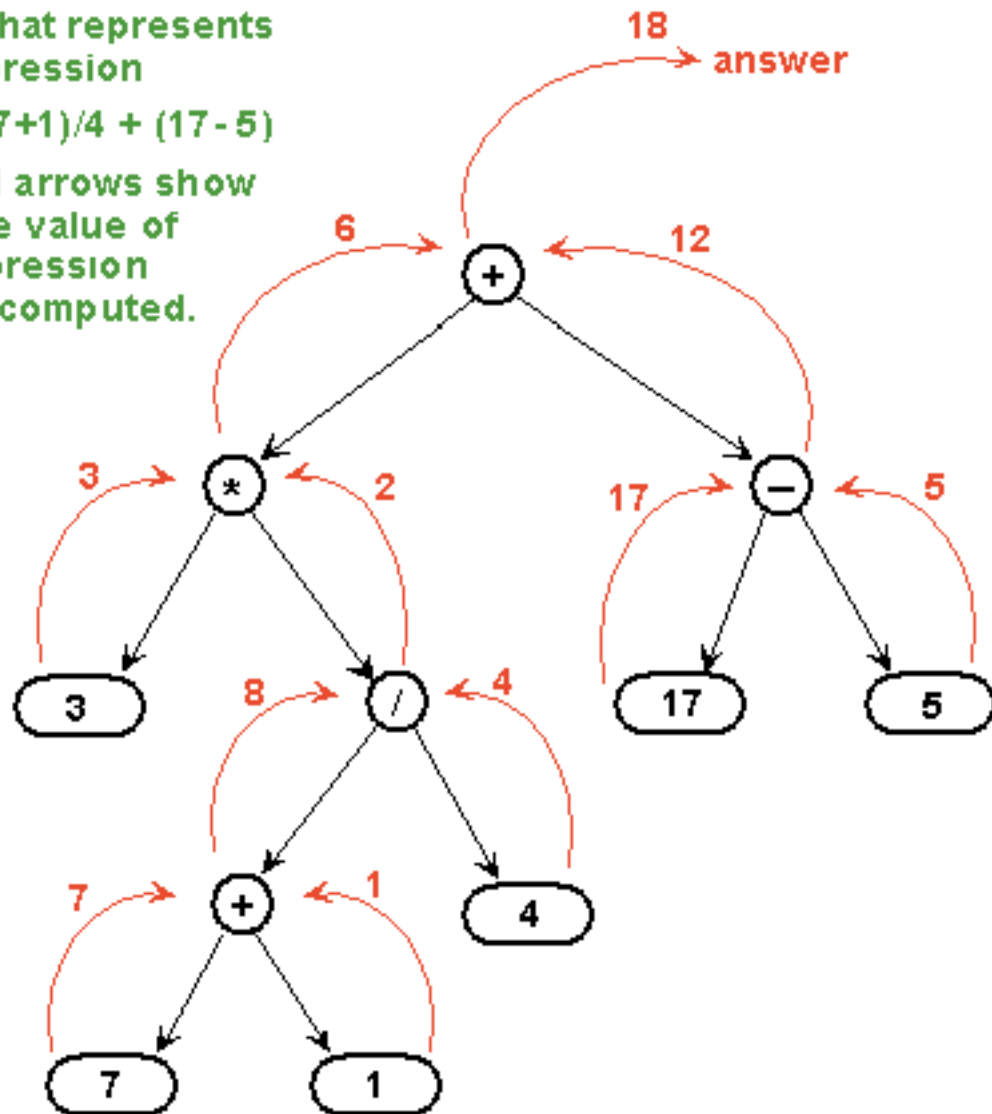
Given an expression tree, it's easy to find the value of the expression that it represents. Each node in the tree has an associated value. If the node is a leaf node, then its value is simply the number that the node contains. If the node contains an operator, then the associated value is computed by 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 a <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();
    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:

Sorry, but your browser
doesn't support Java.

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 END 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. Behind you are all the basics of programming. Ahead are advanced concepts, techniques, and applications that can take a lifetime to explore and master. I hope that you have enjoyed the journey!

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](#)]

Appendix 1

From Java to C++

WHEN I WROTE THE FIRST VERSION OF THESE NOTES in 1996, Java was still a very new programming language. Although it had already caused a lot of excitement, it's long-term prospects were not entirely clear. Now, in the year 2000, I think it is clear that Java is an important language and will remain so for the long term. However, it is true at least for now that most "serious" programming is done in C and C++. Fortunately, these languages share a lot of features with Java. The older language, C, has no object-oriented features. C++ is a much larger language, which extends C with classes, objects, and other features. This chapter serves as a brief introduction to C++ for someone who already knows Java. The coverage here is very incomplete, and is meant only as a starting point for learning about C++.

You'll find that a lot of the basics ("programming in the small") are almost identical in Java and C++. However, both the programming philosophy and the large-scale structure of programs ("programming in the large") in C++ are quite different from Java.

In the first and second editions of this text, this material on C++ was a full-fledged chapter in the text proper, rather than an appendix. It changed very little between the first and second editions, and is completely unchanged (except for section titles) between the second edition and the third.

Contents of this Appendix:

- Section 1: [C++ Programming Fundamentals](#)
 - Section 2: [Pointers and Arrays in C++](#)
 - Section 3: [Classes and Objects in C++](#)
-

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

Appendix 1, Section 1

C++ Programming Fundamentals

WHEN I DECIDED TO TEACH COMPUTER SCIENCE 124 with Java rather than C++, it was with the knowledge that most students would still have to learn C++ eventually. This is because C++ is still the major programming language for professional programmers. In addition, it has a number of language features that Java lacks and that computer science students should be familiar with. However, it was my feeling that Java would be a superior language for an introductory course, and that in any case much of what could be covered in an introductory Java programming course would also apply to C++. After two years of teaching Java and teaching C++ to students who had studied Java as their first programming language, I think I can say that my experience has confirmed or even exceeded expectations. I might even go so far as to advise someone whose only goal is to learn C++ to start with Java.

Before beginning, I should note that recently, after eight years of work, a new standard version of C++ has been released. This new version makes several important changes to the language. It also adds a large, standard collection of classes called the **Standard Template Library**. In this chapter, I am talking about the **older version of C++**. For example, I say below that C++ has no standard String class. However, in the new version, the Standard Template Library does provide such a class. (Of course, until everyone actually makes the switch to the new version, the C++ String class is still not really standardized.)

But before discussing C++, let me mention some of the things that it does not have (even in the new version). First of all, there is no platform-independent GUI support in C++. A typical first course in C++ might not even mention windows, graphics, or events. In addition, standard C++ does not have built-in support for threads or asynchronous programming. Nor does it have a standard interface for network programming. Of course, classes can be written to provide these features in C++. But since they are not standardized, such classes tend to be platform-dependent so that what you learn about one type of computer doesn't automatically apply to other computers. They also tend to be more complicated than the versions available in Java. By using Java, I was able to include these important features of modern computing environments in a natural way.

But enough propaganda about Java. The rest of this chapter is a very brief overview of C++.

Program Structure

The first thing to know about C++ is that you can write entire, complicated programs without ever using classes or objects. Whereas in Java everything, even a program's `main()` routine, must be contained inside classes, C++ allows free-standing subroutines and variable declarations. The `main()` routine, in particular, cannot be a member of a class.

C++ variables and subroutines that are defined outside classes are similar to static variables and subroutines in Java. For example, they exist for the whole time that the program is running. In C++, such variables and routines are said to be **global** or to have **global scope**. They can be used "globally" inside any subroutine and even in methods that belong to classes.

However, no variable, class, or subroutine can be use in a C++ program unless it has been declared earlier in the same file. (This is different from Java, which allows you to use a variable or subroutine that is not declared until later in the program; of course, Java even allows you to use classes that are defined in other files, and it will go off and search for them when necessary.) This might seem to require the entire program to be in one big file, with the `main()` routine at the end, but that is not the case. To avoid this, C++ uses **header files**. A header file contains declarations of variables, subroutines, and classes that are actually defined in other files. C++ distinguishes carefully between **declarations** and **definitions**. The declaration of

a subroutine, for example, gives only its name, its return type, and its parameter list. The definition of a subroutine also includes the code that defines the routine. A subroutine or variable must be declared before it is used in a source code file, but its definition can come later or even in another file. A header file contains only the declaration, which gives enough information to the compiler for it to check whether the subroutine or variable is being used legally.

There are many standard header files that declare useful subroutines and classes. For example, the header file `math.h` defines mathematical functions such as `sin(x)` and `sqrt(x)`. (By convention, the name of a header file ends with ".h".) Another standard header file, `iostream.h`, defines stream classes that can be used for doing input and output. To get access to the declarations in a header file, a program must **include** the header file. This is done with a `#include` statement at the beginning of the program. For example, a program that starts with the statements

```
#include <math.h>
#include <iostream.h>
```

will be able to use the mathematical functions and the stream classes. The `#include` statement is very similar to the `import` statement in Java. For example, saying `#include <iostream.h>` in C++ is similar to saying `import java.io.*` in Java. It's often a chore just determining which header files your program needs.

When you write a large program in C++ and you want to break it up into several pieces, you will generally have to write two files for each piece: one file that actually defines some subroutines and classes, plus a header file that declares the same subroutines and classes. Other files that need access to those subroutines and classes must `#include` the header file to get access to them.

Types and Variables

C++ has standard types named `short`, `int`, `long`, `float`, `double`, and `char` that are similar to Java's primitive types with the same name. In C++, however, the exact meaning of these types is not completely standardized. I can't tell you the range of legal values for `ints`, for example, because the legal range is different on different systems. There is no standard `byte` data type, but `char` really plays the same role, since a `char` in C++ is considered to be simply an 8-bit integer.

C++ does not really have a standard `String` type that is equivalent to the `String` type in Java. C++ does have some support for strings, but they are more difficult to use and they require a knowledge of pointers and arrays. I'll discuss them in the next section. It is possible to write a `String` class in C++ that is very similar to Java's class, and most C++ programmers would probably have access to such a class by `#including` a `String.h` header file. However, you can't depend on this since it's not a standard part of the language.

Most versions of C++ (including the new standard) use `bool` as the name of the boolean type, and the names `true` and `false` are used for the values of type `bool`. However, in C++, `true` and `false` are names for the integers 1 and 0. In fact, an integer can be used wherever a `bool` value is expected, such as in an `if` statement. Any non-zero integer is considered to be `true`, while zero is `false`. So, to test whether the integer `N` is non-zero, you could say `"if (N)..."`

I should note that there is a lot more to say about types in C++, since C++ has several different ways to define new types. Like Java, C++ has class types and array types. But it also has things called pointer types, structs, enumerated types, and function types. The later three types of types, I won't even mention again.

Variables are declared and initialized in C++ in much the same way as in Java. C++ has global variables (declared outside subroutines and classes), instance variables in classes (both static and non-static), and local variables (declared inside subroutines or methods). As in Java, local variables can be declared at any point in a routine, and are valid only for the block in which they are declared. One big difference is that in

C++, a variable can actually contain an object, whereas in Java, a variable can only contain a reference to an object. I'll discuss this further in [Section 3](#).

Control Structures, Assignment Statements, and Expressions

Fortunately, there is very little to say about control structures in C++. Except for a few minor details that you will probably never encounter in practice, they are the same as in Java. That is, C++ has `if` statements, `for` statements, `while` loops, `do` loops, and `switch` statements that work the same way as their counterparts in Java. Also, you can use `break` statements and `continue` statements in C++, just as in Java.

The same goes for assignment statements and expressions. All the operators are the same: `+`, `-`, `*`, `/`, `%`, `++`, `&&`, `||`, `=`, `+=`, and so on. One difference, already noted above, is that the mathematical functions are not contained in a `Math` class. Thus, instead of saying `y = Math.sqrt(x)`, you would say simply `y = sqrt(x)` (On the other hand, if you want to use the `sqrt()` function in C++, you have to `#include` the header file `math.h` at the beginning of your program. Whereas the `Math` class is automatically imported into every Java program, the `math.h` header file is not automatically included in a C++ program.)

Input and Output

Like Java, C++ has a standard input stream and a standard output stream for doing console input/output. The C++ versions are actually much more well-developed than the Java versions, but the syntax for using them is a bit strange. The standard input stream is called `cin`, and the standard output stream is called `cout`. (The "c" in these names stands for "console.") To use these streams in a program, you must `#include` the header file `iostream.h`.

The syntax for reading a value from the standard input stream into a variable named `x` is:

```
cin >> x;
```

(The double-arrow operation is meant to indicate the direction of flow of information.) Output is similar, except that the direction of the arrows is reversed. For example,

```
cout << x;
```

will output the value of the variable `x` to the console, and

```
cout << "Hello World!\n";
```

will output the message "Hello World!". The new line character, `\n`, outputs a carriage return at the end of the message. You can string several output items into a single statement, like this:

```
cout << "The square root of " << x << " is " << sqrt(x);
```

You can do the same thing with `cin`.

The standard I/O streams are instances of general stream classes called `ostream` and `istream`, which are analogous to Java's `PrintStream` and to my `TextInputStream`, respectively.

Subroutines

Subroutine definitions look pretty much the same in C++ as in Java, except of course that they are not necessarily found inside classes. The access modifiers `public`, `private`, and `protected` cannot be used outside classes. The `static` modifier can be used both inside and outside classes, but outside a class, it has a completely different meaning than it does inside a class. (It means that the subroutine is for use only in the file in which it is defined. The word `static` is grossly overused in C++, with several very different meanings in different contexts.)

Subroutine call statements and function invocations are also just about identical in Java and in C++.

Every program must have exactly one `main()` subroutine, and just as in Java, the system runs the program by calling its `main()` routine. The exact form of the main routine is not entirely standardized, and often several forms are acceptable. In some cases, for example, the main routine has return type `void`, while in other cases, it has return type `int`. (The `int` that is returned by the main routine is sent back to the system -- which called that routine -- as a status code to indicate whether the program completed successfully or ended in error.) On a given system, either or both of these formats might be acceptable.

Exception Handling

Although exception handling is not really a "programming fundamental," I mention it in this section because it is another area in which Java and C++ are very similar. C++ uses `try`, `catch`, and `throw` in the same way as C++. However, C++ does not allow a `finally` clause in a `try` statement, and it does not have the large number of predefined exception classes that Java has. Since exception handling is a recent addition to C++, it is still not completely well-integrated into the language. Many of the standard subroutines, for example, were written before exception handling was introduced, and they use older, alternative ways to deal with errors.

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

Appendix 1, Section 2

Pointers and Arrays in C++

A POINTER IS JUST THE ADDRESS OF SOME location in memory. In Java, pointers play an important role behind the scenes in the form of references to objects. A Java variable of object type stores a reference to an object, which is just a pointer giving the address of that object in memory. When you use an object variable in a program, the computer automatically follows the pointer stored in that variable in order to find the actual object in memory. Since all this happens automatically, behind the scenes, you really don't have to worry much about the fact that objects are referenced through pointers.

In C++, everything about pointers becomes explicit. Some variables can store pointers, but only variables that are of special **pointer types**. When you use a pointer variable, the computer does not follow the pointer automatically -- if you want the computer to follow the pointer, you have to use a special notation to tell it to do so. You can create a new object for a pointer variable to point to, using a "new" operator that is similar to the new operator in Java. But where Java will dispose of such objects automatically when you are done with them, in C++ you are responsible for keeping track of objects created with the new operator and disposing of them when they are no longer needed. Another operator, named `delete` is provided for doing this. All-in-all, working with explicit pointers is fairly difficult and error-prone, and it is generally considered to be one of the advanced aspects of programming. Unfortunately, in C++ there are some pretty basic things that you can't do without using pointers (not unless you use other advanced features of the language or use a class that someone else has written for you).

Suppose that `T` is the name of a type in C++. Then there is a pointer type named `T*` which represents variables that can hold pointers to objects of type `T`. C++ allows pointers to values of any type, not just objects. For example, there is a type `int*` that represents pointers to integers. A variable of type `int*` holds the address of a location in memory; an integer value is stored at that location. Suppose that `T` is some type and that `ptr` is a variable of type `T*`. When you use `ptr` in a program, you are referring to the actual pointer, that is, the memory address that is contained in the variable `ptr`. To refer to the value that is stored at that address, you would use the notation `*ptr`. C++ uses the predefined constant `NULL` -- all upper case -- as the value of a pointer that doesn't point to anything. It is an error to refer to `*ptr` if the value of `ptr` is `NULL`.

The notation for defining a variable, `ptr`, of type `T*` is:

```
T *ptr;
```

Thus, to declare `iptr` as a variable of type pointer-to-int, you would say:

```
int *iptr;
```

The notation "`T *ptr`" is meant to suggest that the value of `*ptr` is of type `T`. But note that the variable that is being declared is `ptr`, not `*ptr`.

The type of a subroutine parameter can be a pointer type. For example, here is a subroutine that will swap the integer values stored in two memory locations. Its parameters are of type pointer-to-int:

```
void swap(int *x, int *y) {
    int temp;
    temp = *x; // copy the value pointed to by x into temp
    *x = *y;   // copy the value pointed to by y into the
              // location pointed to by x
    *y = temp; // copy the value in temp into the location
              // pointed to by y
}
```

At the end of this subroutine, `x` and `y` still point to the same memory locations, but the values in those locations have been swapped.

This becomes more interesting once you know about the **unary address operator**, which is written as `&`. Suppose that `x` is any variable. Then `&x` is a notation that stands for the memory address of `x`. That is, `&x` is a pointer that points

to x. Consider the following C++ program segment:

```
int x = 7;    // x is an integer variable with initial value 7
int *iptr;   // iptr is a variable that can hold a pointer to an int
iptr = &x;   // iptr now holds a pointer to x, so that now
             //      x and *iptr refer to the very same memory location
*iptr += 3;  // adds 3 to the value stored in location *iptr
cout << x;   // print out the value of x, which is 10
```

Because `*iptr` and `x` refer to the same memory location, adding 3 to `*iptr` is really the same as adding 3 to the value of `x`. So the output value of `x` in this example is `7 + 3`.

Suppose that we combine the address operator with the `swap ()` routine defined above. Consider this example:

```
int a,b;
a = 17;
b = 42;
swap(&a, &b);
```

The subroutine call is legal because `&a` and `&b` are pointers to integers. That is, they are of type `int*`, and that is the same type as the formal parameters of the subroutine. The end result of this example is that the original values of `a` and `b` are swapped. So the final value of `a` is 42, and the final value of `b` is 17. Believe it or not, using pointers and addresses like this is a very common way to write subroutines that can alter the values of variables. (In the C programming language, it's the only way. C++ provides a somewhat more slightly alternative, which is discussed below. Note that in Java, it is impossible to write a subroutine that performs the same task as `swap ()`, since there is no such thing as a pointer-to-int in Java. Of course, you can do similar things with objects and references to objects, in Java as well as in C++.)

Pointers are probably used most often with classes, but I will put off that discussion to the next section.

References and Passing Parameters by Reference

In C++, it is possible to write a subroutine that can change the value stored in a variable, when that variable is passed to the subroutine as an actual parameter. Here is a variation of the `swap` routine that does this:

```
void swap(int& x, int& y) {
    int temp = x;
    x = y;
    y = temp;
}
```

The type `int&` which is specified in this subroutine for the formal parameters `x` and `y` is called a **reference type**. If `T` is any type in C++, then you can form a reference type, `T&`. You can actually declare variables using reference types, but I have never seen any practical use for them except for formal parameters and occasionally as the return type of a function. When a formal parameter is declared using a reference type, the parameter is said to be **passed by reference**. The effect of passing a parameter by reference is to allow the subroutine to change the value of the actual parameter. Thus, if the subroutine `swap ()` is defined as above, using pass-by-reference, then the subroutine call statement `swap (a , b)` will interchange the values stored in the variables `a` and `b`. On the other hand, consider the subroutine

```
void failsToSwap(int x, int y) {
    int temp = x;
    x = y;
    y = temp;
}
```

In this case, a subroutine call statement `failsToSwap (a , b)` will have no effect at all on the values stored in `a` and `b`. The values of the actual parameters `a` and `b` are **copied into the formal parameters `x` and `y` when the subroutine is called**. The subroutine interchanges the values of `x` and `y`, but this has no effect on what's stored in `a` and `b`. In the `failsToSwap ()` subroutine, the parameters are said to be **passed by value**.

When a parameter is passed by reference, then what's passed to the subroutine is really a reference, or pointer, back

to the actual parameter. The effect is similar to what was done in the first version of `swap()`, which passed pointers explicitly. But in pass-by-reference, the pointers are handled implicitly.

In Java, all parameters are passed by value, and so there is no way to write a subroutine that will swap the values stored in two integer variables. More generally, in Java, no subroutine can ever change the value stored in a variable that is passed to the subroutine as an actual parameter. This is kind of a shock to programmers who are used to other languages. (Be careful: when an object variable is passed as a parameter in Java, the value stored in the variable is a reference to an object. The values of instance variables in that object can be changed by the subroutine, even though the reference stored in the variable cannot be changed. When the subroutine ends, the variable will still refer to the same location in memory, but the values in the object stored at that location might have changed. In a sense, then, objects are passed by reference in Java, even though variables are not.)

Arrays

Arrays in C++ are superficially similar to arrays in Java. An array is just an indexed list of values, where the index goes from 0 up to some maximum. But in C++, an array variable is considered to be nothing more than a pointer to the first element in the array. In fact, array variables and pointer variables can be used interchangeably. For example, if `list` is an array variable, then you can use either `list[0]` or `*list` to refer to the first item in the array. And you can use an assignment statement to copy the value of an array variable into a pointer variable.

Arrays can be created in C++ using the `new` operator, just as in Java. For example, to create an array of 100 ints, you could say either

```
int list[] = new int[100];
```

or

```
int *list = new int[100];
```

(Note, however, that the notation for declaring an array variable is "int list[]", not "int[] list". In Java, either notation would be allowed, but I prefer the latter.)

It is also possible to create arrays without using the `new` operator:

```
int list[100]; // declares an array of 100 ints
```

The main difference is that arrays created with the `new` operator should eventually be disposed of with the `delete` operator, in order to avoid memory leaks. Arrays created with the second notation do not have to be -- and in fact cannot be -- deleted. Most beginning C++ programmers would probably use arrays for quite a while before they even learn about the `new` operator.

Although an array in C++ has some fixed length, the computer does not keep track of that length in any way, and there is nothing to stop you from trying to refer to array locations that don't even exist. Thus, arrays -- like pointers in general -- are a common source of bugs in C++ programs.

Strings as Arrays of Characters

Strings in C++ are considered to be arrays of characters. A string, therefore, has type `char[]`. Since array types and pointer types are considered to be equivalent, we could also consider a string to be of type `char*`, that is pointer to `char`.

Since a string in C++ is just an array of characters, strings can be manipulated using array notation. For example, if `str` is a string variable, then `str[i]` refers to the *i*-th character in the string. You can change one of the characters in a string by using an assignment statement to the appropriate array element. For example, if `str` is the string "bit", then the assignment statement `str[1]='e'` changes the value of `str` to "bet". (In Java, the *i*-th character of a string `str` is referred to as `str.charAt(i)`, and there is no way to change the characters in a string.)

In C++, there is no `String` class, and strings are not objects. So there are no class methods for operating on strings.

Instead, a large number of subroutines for working with strings are available through a standard header file, `string.h`.

Since strings are arrays and since the computer does not keep track of the lengths of arrays, the computer does not know how long a string is. However, by convention, strings in C++ are assumed to end with a null character (the character with ASCII code zero). The null character is not really considered to be part of the string. It's there just as an end-of-string marker. This convention is used when a string variable is initialized. For example, if you say

```
char *greeting = "Hello World";
```

the computer creates an array of 12 characters, not just 11, to hold the string. The characters in `greeting[0]` through `greeting[10]` are the characters 'H' through 'd' in the string that you've specified. The computer adds a null character in the last location, `greeting[11]`, to mark the end of the string.

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

Appendix 1, Section 3

Classes and Objects in C++

C++ IS BASED ON AN EARLIER PROGRAMMING LANGUAGE called C. The main distinguishing feature of C++ is its support for object-oriented programming with classes and objects. (Although, in fact, C++ adds many other features to C as well.)

Classes in Java are modeled on classes in C++, but with many differences. The most obvious of these is due to the distinction between declaration and definition in C++. (See [Section 1](#).) In C++, a class has two distinct components, which can be -- and most often are -- placed in different files. First of all, there is the class declaration, which is similar to a class in Java except that it omits the definitions of the methods. The class declaration contains only what are called **prototypes** for the methods. It also declares any variables that belong to the class. The class declaration is generally placed in a header file so that it can be used easily by other parts of the program. The second component of the class consists of the actual definitions of the class's methods. These definitions are separate from the class declaration, and usually occur in a separate file.

Here, for example, is a simple C++ class declaration:

```
class SimpleCalc {

    // A class for computing some statistics about
    // a list of numbers; the numbers are entered
    // into the data set one-by-one by calling the
    // method enter().

    int count;        // Number of items that have been entered.
    double sum;        // The sum of items that have been entered.
    double sqSum;      // The sum of the squares of all the items
                       // that have been entered so far.

public: // The following members of the class are
       // visible from outside the class.

    SimpleCalc();      // Constructor

    void enter(double dataItem); // Adds dataItem to the data set.

    int getCount();    // Returns the number of items that
                       // have been entered into the data set.

    double getMean();  // Gets the average of the items that
                       // have been entered so far.

    double getSD();    // Gets the standard deviation.

    void clear();       // Clears the data set, resets count to zero.

}; // end of declaration of class SimpleCalc
```

This should look mostly familiar from Java, except for the omission of the bodies of the methods. The syntax for the "public" modifier is a bit different. Instead of being applied to individual items, the modifiers `public`, `private` and `protected` apply to groups of items. In the above example, all the methods that follow "public:" are public. Items at the beginning of a class, before any access modifier is specified, are `private` by default. (Also note the semicolon at the end of the class declaration, which is absolutely required. Making this semicolon optional was one of the things that the designers of Java did right.)

You can apply the "static" modifier to individual methods and variables in a class, with the same meaning as in Java. Unfortunately, neither static nor instance variables can be given initial values in a class declaration. (This is one of Java's more substantial improvements on C++.) Instance variables should be initialized in a constructor. There is a way to do initialization of static variables, but it has to be done outside the class declaration, as part of the definition of the class.

By the way, in addition to constructors, a C++ class can contain a special method called a **destructor**. Just as a constructor is called automatically by the system when an object is first created, a destructor is called by the system when the object is destroyed. For an object created with the new operator, this happens when the object is explicitly deleted with the delete operator. For objects that are not created with the new operator, the system is responsible for making sure that the object is destroyed at the appropriate time.

The purpose of a destructor is to do any cleanup that is necessary when the object is destroyed (beyond reclaiming the memory that is occupied by the object, which is being done in any case). For example, the destructor might close files that were opened in the constructor. Or if the constructor created an object with the new operator, the destructor can apply the delete operator to that object.

The name for a destructor is the same as the name of the class, preceded by the character '~'. A destructor for the SimpleCalc class would be named ~SimpleCalc. (Java does not have destructors. However, a class in Java can contain a "finalizer()" method which serves a similar purpose.)

Now, let's say that the declaration of the class SimpleCalc is in a header file named SimpleCalc.h. To complete the definition of the class, you would write another file -- probably called SimpleCalc.cpp or SimpleCalc.cc -- to give the definition of each of the methods in the class. Here's what would go in that file:

```
#include "SimpleCalc.h"
#include "math.h"

SimpleCalc::SimpleCalc() {
    count = 0; // initialize the instance variables
    sum = 0;
    sqSum = 0;
}

void SimpleCalc::enter(double dataItem) {
    count++;
    sum += dataItem;
    sqSum += dataItem * dataItem;
}

int SimpleCalc::getCount() {
    return count;
}

double SimpleCalc::getMean() {
    return sum / count;
}

double SimpleCalc::getSD() {
    double mean = getMean();
    return sqrt( (1.0/count) * (sqSum - count * mean * mean) );
}

void SimpleCalc::clear() {
    count = 0;
    sum = 0;
    sqSum = 0;
}
```

The first line of the file might surprise you. To compile the definition of the class methods, the compiler must know the complete declaration of the class. However, the C++ compiler never goes out and looks for information -- if it needs to know something that's in another file, then that file must be explicitly `#included`. So in this case, the header file `SimpleCalc.h`, which contains the declaration of the class, must be included.

The second line, which `#includes` `math.h`, is required since `math.h` declares the `sqrt` function, which is used later in the file.

The rest of the file consists of definitions for the six methods of `SimpleCalc`, counting the constructor. The only surprise here is that the name of each method is preceded by `"SimpleCalc::"`. This identifies the definition as a method in the class `SimpleCalc`, rather than an independent subroutine. (The same notation, by the way, is used to access static members of a class. If `SimpleCalc` had a static variable named `data`, then outside the class it would be called `SimpleCalc::data`.)

To finish up this example, here is a main program that uses the class, `SimpleCalc`. This would be in yet another file, perhaps named `main.cpp`. Note that it includes the header file `SimpleCalc.h`, which declares the class, but it does not include the definitions of the methods in the class, which are defined in the file `SimpleCalc.cpp`. These definitions are inside the "black box," and the main program doesn't need to know them. This program also uses the standard `iostream.h` header file to get access to the standard input and output streams, `cout` and `cin`.

```
#include "SimpleCalc.h"
#include "iostream.h"

main() { // Main routine for the program.
        // Reads a list of numbers input by the user and
        // prints some simple statistics about the data.

    SimpleCalc calc; // A SimpleCalc object to keep track
                     // of the data and compute the statistics.

    double x; // A number entered by the user

    cout << "Enter some numbers.\n";
    cout << "Enter 0 to end.\n";

    do {
        cout << "    ? ";
        cin >> x;
        if (x != 0)
            calc.enter(x);
    } while (x != 0);

    cout << "You entered " << calc.getCount() << " numbers.\n";
    cout << "The average is " << calc.getMean() << " .\n";
    cout << "The standard deviation is " << calc.getSD() << " .\n";

}
```

One important note is that the `SimpleCalc` object is created without using the `new` operator. In C++, a variable of object type, such as `calc` in this program, holds an object and not just a reference to an object. When such a variable is declared, an object is constructed and is stored in the variable. If the constructor requires parameters, they are provided as part of the variable declaration. For example, if the constructor in the class `SimpleCalc` required a parameter of type `int`, the declaration of `calc` would take this form:

```
SimpleCalc calc(10);
```

This is very different from Java. To get behavior more similar to what happens in Java, you have to use pointers to objects. For example, you could declare a variable `pcalc` that can hold a pointer to an object of type

SimpleCalc:

```
SimpleCalc *pcalc;
```

In this case, just as in Java, this does not automatically create an object for `pcalc` to point to. For that you need the new operator:

```
pcalc = new SimpleCalc();
```

One odd thing that happens in C++ is that when you use a pointer to refer to an object, the notation for referring to instance variables and methods in the object also changes. Since `*pcalc` is the name of the object, you could say "`(*pcalc).clear()`" to refer to the `clear()` method in the object `*pcalc`. (The parentheses are necessary because `*pcalc.clear()` would be taken to mean `*(pcalc.clear())`. The `'.'` operation has higher precedence than the `'*'` operation.) The notation `(*pcalc).clear()` is rarely used, however, because the alternative notation "`pcalc->clear()`" is defined to mean the same thing. The operator `"->"` means "first follow the pointer to find an object, then access a member of the object."

To have real object-oriented programming, of course, you have to have inheritance and polymorphism. Inheritance is easy. Just as in Java, you can declare one class to be a subclass of another. For example, to declare `ComplexCalc` to be a so-called "public" subclass of `SimpleCalc`, you would say:

```
class ComplexCalc : public SimpleCalc {
    .
    . // (additions and modification to stuff
    . //   that is inherited from SimpleCalc)
    .
}
```

This would be equivalent to saying in Java that `ComplexCalc` extends `SimpleCalc`. Just as in Java, `ComplexCalc` inherits everything that is contained in the definition of `SimpleCalc` and it can modify and add to what it inherits. (There are also "private" and "protected" subclasses, but I won't try to explain them here.)

Polymorphism, unfortunately, is a bit murky in C++. First of all, polymorphism works only with pointer variables and variables of reference type, never with variables that contain actual objects. A pointer variable can point to objects belonging to some specified class or to a subclass of that class. So, you would expect a subroutine call statement of the form `ptr->doSomething()` to be polymorphic. That is, you would expect that the `doSomething` method that is called will depend on the type of object that `ptr` is actually pointing to at the time the statement is executed. However, this will happen only for so-called **virtual methods**. You can declare a method to be virtual by adding the modifier `virtual` to the declaration of the method. What you are saying when you do this is that the method is meant to be polymorphic. If you think this is confusing, don't worry too much about it -- I've talked to people who had studied C++ for quite a while and who still could not correctly explain exactly what `virtual` means. Just remember that if you want to do real object-oriented programming in C++, then you have to use pointers and virtual methods. You might also remember this as one more reason to appreciate Java, where polymorphism is automatic and natural.

This has been just a brief introduction to classes in C++, which have many features that I haven't even mentioned. More generally, C++ is a very large and complex language, which takes a long time to master. I certainly haven't told you enough about it to enable you to write serious programs in C++, but I hope I've given you some of the its general flavor and a sense of how it differs from Java.

End of Appendix 1

[Previous Section](#) | [Appendix Index](#) | [Main Index](#)]

Appendix 2:

Some Notes on Java Programming Environments

EACH TIME I HAVE TAUGHT a course based on this textbook, I have used a different programming environment: CodeWarrior for Macintosh, Visual J++ for Windows, and the JDK for Linux. Of the three development environment environments, the JDK was by far the most successful. CodeWarrior and Visual J++ are examples of **integrated development environments**. That is, they include a text editor, an interactive debugger, a compiler, and tools for managing large programming projects. The problem with such environments in an introductory course is their complexity, which can be overwhelming for a beginner who is having trouble enough learning the basics of programming. The JDK (Java Development Kit) uses a command-line interface, in which the user types commands to compile and run Java programs. Although this interface is a bit alien to students who are used to working with a graphical user interface, I have not run into any students who had difficulty understanding or using it. Another advantage of the JDK is that it is free.

In this appendix, I'll give some information and opinion on Java programming environments for Windows, Macintosh, and Linux/UNIX. I'll concentrate on free software, including the JDK. There is a lot of redundancy among the sections on Windows, Macintosh, and Linux/UNIX. You probably only need to read one of these sections.

But first, let me describe the computing setup that I use in my courses, since that might be of interest to other professors.

[My department](#) has a small computing lab of its own, with a mixture of Linux, Windows, and Macintosh computers. The majority of the computers run Linux. The Linux computers use a graphical ("XDM") login screen and the [KDE desktop](#), which means that they work much the same as Windows or Macintosh from the user's point of view. The lab is not large enough for introductory programming classes, but students can log on to the Linux machines in the lab from any Windows computer on campus. For this, we use X-Win32, from StarNet (www.starnet.com). Using X-Win32, students see the same login screen and desktop as they see when they work on the machines directly in the lab. (Although X-Win32 is fairly expensive for commercial use, StarNet has a very reasonable academic pricing policy.) Student home directories are on a departmental server, and are accessible from anywhere on campus. The Linux distribution that we use, currently SuSE 6.1, provides many tools that we use in introductory programming and other courses, including the JDK, text editors, C++, Lisp, Prolog, and OpenGL (in the form of Mesa).

Sources of Software and Information

If you work in Windows, I suggest that you take a look at burks.brighton.ac.uk. You'll find a large collection of Windows software that is useful for students of computer science. The collection can be purchased on a set of cheap CD's, and it is available on-line at <http://burks.brighton.ac.uk/burks/index.htm>. (Version 4 of this collection includes the second edition of this textbook. It also has a copy of JDK 1.1.6, which is suitable for use with this textbook.)

The home site for Java, which was invented by Sun Microsystems, is java.sun.com. For version 1.1 of the Java Development Kit see <http://java.sun.com/products/jdk/1.1/>. Version 1.2 is at <http://java.sun.com/products/jdk/1.2/>. (This textbook requires JDK version 1.1 or higher. Version 1.2 is a much larger download.)

For information about programming in Java on Macintosh computers, see <http://developer.apple.com/java/>.

A good general source for Java programming information is www.gamelan.com.

Text Editors

Before you start writing programs, 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.
- Autoindentation. 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.

For Linux and UNIX, I recommend *nedit* as a nice editor and one that will be comfortable for people who are used to GUI programs for Windows and Macintosh. 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/> for many UNIX platforms. Features such as syntax coloring and autoindentation are not turned on by default. You can configure them in the Options menu. Use the "Save Options" command to make the configuration permanent.

For Macintosh, I recommend BBEdit Lite from [Bare Bones Software](http://www.barebones.com). BBEdit Lite is a free version of the excellent text editor, BBEdit. It can be downloaded at http://www.barebones.com/free/bbedit_lite.html. You can also ftp to <ftp://ftp.barebones.com/pub/freeware/> and download [BBEdit Lite 4.6.hqx](#). This version requires System 7 or higher. (You will need Stuffit Expander to extract the application from the hqx file. You probably already have Stuffit Expander, since it comes with most Web browsers. If not, you can download it from www.aladdinsys.com.) You might also want to download [PopupFuncs 2.8.2.hqx](#) from the same ftp directory. PopupFuncs can be used to add pop-up function menus to BBEdit Lite. Unfortunately, the free version of BBEdit does not do syntax coloring.

For Windows, you might consider the free text editor SynEdit. It can be downloaded at www.mkidesign.com. This program is distributed as a "zip" file, so you will need a program such as WinZip to extract it. (You can download WinZip from www.winzip.com.) I have also used Editeur, but it is shareware rather than freeware. (However, I haven't used Windows enough to have a strong recommendation.)

Java for Windows 95 or NT (or later)

If you use the JDK on Windows 95/98, you will be working in DOS command windows, with a command-line interface. A utility called "doskey", which is included in the standard Windows installation, will make your life easier. To install it, add the line

doskey

to your `autoexec.bat` file. The change will take effect when you restart your computer. Once doskey is installed, when you are working in a DOS command window, you can use the up arrow key to go back to previous commands that you have typed in that window. Once the command is retrieved, you can edit it, if you want. Then press return to issue the command again. Since you will often find yourself issuing the same commands over and over as you compile and run your programs, this can save you a lot of typing.

Note: To edit your `autoexec.bat` file, select the "Run" command from the "Start" menu. Enter the command **sysedit** in the dialog that pops up, and press return. This will open a window where you can edit system configuration files, including `autoexec.bat`. Make your change, such as adding the "doskey" command to this file, and save your changes.

You can download version 1.1 of the JDK for Windows at <http://java.sun.com/products/jdk/1.1/>. You might also want to pick up a copy of the documentation there. JDK 1.1.6 is also available on the Burks site (<http://burks.brighton.ac.uk/burks/index.htm>). You should be able to use a later version of the JDK with this textbook, but not an earlier one. Follow the installation instructions. This will include modifying your `PATH` to include the directory that contains the java compiler and interpreter. (On Windows 95/98, you will have to edit your `autoexec.bat` file and restart your computer to do this. See the above note about editing `autoexec.bat`. You have to add the path for the bin directory of the java installation to the end of the line that starts "SET PATH=". If there is no such line in the `autoexec.bat` file, add one. The path to the bin directory will be something like `C:\jdk1.1.6\bin` if you used the default JDK installation. The line in the `autoexec.bat` file will have the form: "SET PATH=%PATH%;maybe-other-paths;C:\jdk.1.6\bin".)

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 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 browser page.

To use the JDK, you will have to work in a DOS window, using a command-line interface. Open a DOS Window and change to the directory that contains your Java source code files. (Tip: In Windows 95/98, open a directory window for the directory, then select the "Run" command from the "Start" menu, enter "command" in the dialog window that pops up, and press return. A DOS window will open that is all set for working in the directory shown in the directory window.)

The "javac" command is used for compiling Java source code files. For example, to compile the Java source code file named `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 a DOS 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 Windows NT only, but not Windows 95/98, you can save the errors in a file which you can view later in a text editor. Use the command such as

```
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.)

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 JDK 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".)

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 JDK 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 DOS window. This can be useful for debugging your applet.

Of course, it's also possible to open the HTML file in a Web browser, such as Netscape or Internet Explorer. One problem with this is that if you make changes to the applet, you 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 reloaded.

Java for Macintosh

The Apple Computer company has a free Software Development Kit (SDK) for Java. This SDK includes Apple's versions of the compiler from the JDK. To use the SDK, you also need to have a copy of the MRJ (Macintosh Runtime for Java) installed on your computer. The MRJ includes a Java interpreter for running Java applications and applets. If you have a recent version of the MacOS, the MRJ might already be installed. If not, you can download it from Apple's Java Web site at <http://www.apple.com/java/>. The SDK can be downloaded from the FTP directory at

ftp://ftp.apple.com/developer/Development_Kits/

At the time I am writing, the latest version is [MRJ SDK 2.2 Install.sit.hqx](#). For more links and full information on developing Java programs on Macintosh, see <http://developer.apple.com/java/>.

You'll need access to three programs from the MRJ and SDK. I suggest that you make aliases to these programs and leave the aliases on your desktop. (To make an alias, click once on the program icon to hilite it. Choose the "Make Alias" command from the file menu. This will create an icon for the alias. Drag this icon onto your desktop.) The programs you need are:

- **Apple Applet Runner**. You will find this in the "Mac OS Runtime For Java" folder inside a sub-folder named "Apple Applet Runner".
- **javac**. This is part of the SDK. It is in a folder called "JDK Tools" which is itself inside a folder called "Tools".
- **JBindary**. This is also part of the SDK. You'll find it inside a folder called "JBindary".

Make a folder to hold your Java programs. (You might want to have a different folder for each program that you write.) Create your program with a text editor, or copy the program you want to compile into your program folder. 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 folder with the main program file that uses it.

If you have downloaded a copy of this textbook, you can simply copy the files you need from the [source](#) folder that is part of the download. If you haven't downloaded the textbook, you can open the source file in a Web browser and 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 hilite 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 browser page.

To compile a Java source code file drag the file onto the **javac** program icon (or an alias for this program). If the file you are compiling depends on other source code files, select them all and drag them all onto the **javac** icon. For example, if your program uses `TextIO`, you should drag both `TextIO.java` and the source file for the main program onto the **javac** icon. (To select multiple files, hold down the Shift key as you click on the second, third, ... files.) A dialog window will open. It should show the names of the source code files in a box on the upper left. Click the "Do Javac" button in this window. This will do the compilation. A message window will open with messages from the compiler. If your program contains any syntax errors, they will be listed in the message window. You can then edit the file and try to compile it again. Note that you can keep the text editor window and the **javac** window open at the same time. After making changes to the program with the text editor, save your work and hit the "Do Javac" button again.

Once you have the compiled class files, you can run your program. If the program is a stand-alone application, with a `main()` routine, you can run it with **JBindary**. Drag the compiled class file that contains the `main()` routine onto the **JBindary** icon (or an alias). Even if the main program depends on other classes, you only have to drag one file onto the icon. (However, the other class files must be in the same directory with the main class file.) A window should open that shows the name of your main class in a box at the upper left. This window also has two pop-up menus for "redirecting" Standard Input and Standard Output. For programs that use `TextIO`, make sure that both of these pop-up windows are set to the "Message Window". Click the "Run" button to run your program. A separate "Message Window" will open for doing standard input and output, if your program uses them.

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>
```

To run the applet, drag the HTML file onto the **Apple Applet Runner** icon (or an alias).

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. These will appear in a separate "Message Window". This can be very useful for debugging.

Java for Linux/UNIX

Hopefully, Java is already installed on your UNIX or Linux system -- or at least is available on your installation disks. For Solaris and some versions of Linux, you can download the JDK from <http://java.sun.com/products/jdk/1.2/>. Version 1.1 of the JDK for Solaris, but not Linux, is available at <http://java.sun.com/products/jdk/1.1/>. You can also check the Blackdown organization at www.blackdown.org. They produced a port of Java to Linux.

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 such as `nedit`, 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 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 browser page.

You will need a command-line interface to work with the JDK commands. If you are working with a graphical user interface, open an `xterm` (or other command-line window). Of course, you need the GUI to run Java applets. Change to the directory that contains your Java source code files.

The "javac" command is used for compiling Java source code files. For example, to compile the Java source code file named `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. You should have some way of scrolling back to see the first errors in the list. 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 it in another. 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!

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.)

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 JDK 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".)

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 JDK can be used to view an applet from an HTML file. 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-line window. This can be useful for debugging your applet.

[[Main Index](#)]

Introduction to Programming Using Java, Third 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/), 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`. To use examples that depend on other 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 Visual J++, 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.)

Part 1: Text-oriented Examples from the Text

Many of the sample programs in the text are based on console-style input/output, where the computer and the user type lines 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](#). Most of the standalone programs depend on the `TextIO` class, which is defined in [TextIO.java](#).

- [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 expression without building expression trees, are [SimpleParser1.java](#) and [SimpleParser2.java](#).
-

Part 2: Graphical Examples from the Text

- [GUIDemo.java](#), a simple GUI demo applet from [Section 1.6](#). (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 [SimpleAnimationApplet.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 defines six classes. You won't be able to understand everything in this file until you've read Chapters 6 and 7.
- [HelloWorldApplet.java](#), the utterly basic first sample applet, from [Section 6.1](#).
- [ColoredHelloWorldApplet.java](#), the first sample applet that uses a button, from [Section 6.1](#).
- [ColorChooserApplet.java](#), an applet for investigating RGB and HSB colors. This applet uses some techniques that are not covered until Chapter 7. 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](#).
- [SimpleStamper.java](#), a basic demo of MouseEvents, 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 [KeyboardAnimationApplet.java](#), which uses some advanced techniques.
- [ColoredHelloWorldApplet2.java](#), an applet that introduces a simple layout, consisting of a drawing canvas with a bar of control buttons, from [Section 6.6](#). This applet depends on [ColoredHelloWorldCanvas.java](#).
- [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 first, double buffering is used to implement smooth dragging. From [Section 7.1](#).
- [RubberBand.java](#), a little applet illustrating rubber band cursors, implemented using XOR painting mode, from [Section 7.1](#).
- [SimplePaint3.java](#), an improved paint program that uses XOR mode and an off-screen canvas, from [Section 7.1](#).
- [LayoutDemo.java](#), which demos a variety of layout managers, from [Section 7.2](#).
- [EventDemo.java](#), which demonstrates various GUI components, from [Section 7.3](#).
- [ShapeDrawWithMenu.java](#), an extended version of [ShapeDraw.java](#) that uses a pop-up menu, from [Section 7.3](#).
- [RGBColorChooser.java](#), a simplified version of [ColorChooserApplet.java](#) that lets the user select a color with three scroll bars that control the RGB components, from [Section 7.4](#).
- [SimpleCalculator.java](#), which lets the user do arithmetic operations using TextFields and Buttons, 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 component layout instead of using a layout manager, from [Section 7.4](#).
- [BlinkingHelloWorld1.java](#), an applet that blinks a message when the user clicks on a button, from [Section 7.5](#). This is the first example of using a thread. This applet depends on [ColoredHelloWorldCanvas.java](#).
- [BlinkingHelloWorld2.java](#), an applet that blinks a message when the user clicks on a button and stops when the user clicks again, from [Section 7.5](#). This is the first example of communication between two threads. This applet depends on [ColoredHelloWorldCanvas.java](#).
- [ScrollingHelloWorld.java](#), an applet that scrolls a message, from [Section 7.5](#). This is the first example of using synchronization with the `wait()` and `notify()` methods.
- [RandomColorGrid.java](#), which uses nested and anonymous classes, from [Section 7.6](#).
- [ShapeDrawFrame.java](#), another version of [ShapeDraw](#) that uses a Frame, with a menu bar, instead of an applet. 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.

- [MessageDialog.java](#), a class for displaying modal dialogs that contain a message and one, two, or three buttons. From [Section 7.7](#). The applet [DialogDemoLauncher.java](#) is a button that opens a frame that runs a little demo of the `MessageDialog` class.
- [RandomStringsWithArray.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 710 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](#) and [MessageDialog.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 depends on the file [MessageDialog.java](#). It is described at the end of [Section 10.3](#).
- [URLExampleApplet.java](#), an applet that reads data from a URL, from [Section 10.4](#).
- [ConnectionWindow.java](#), a `Frame` that supports chatting between two users over the network, from [Section 10.5](#). This class depends on [TextReader.java](#).
- [BrokeredChat.java](#), an applet that sets up chat connections that use the previous example, [ConnectionWindow.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. Many of them use rather advanced techniques. By the time you finish the course, you should know enough to read the sources for these applets and hopefully learn something from them.

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 [SimpleAnimationApplet.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 5.5](#). Depends on [MosaicCanvas.java](#) and [RandomBrighten.java](#).
6. [TrackLines.java](#), an applet with lines that track the mouse, from [Section 6.7](#). This applet uses Java 1.0 style event handling.
7. [KaleidaAnimate.java](#), an applet that shows symmetric, kaleidoscope-like animations, from [Section 7.8](#). 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. This applet uses the old-fashioned Java 1.0 style event-handling.

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-GUI.java](#) 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.
- [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.
- [SimpleAnimationApplet.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. [Section 3.7](#) has an example.
- [KeyboardAnimationApplet.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 KeyboardAnimationApplet.

- [Mosaic.java](#) which let's you write programs that work with a window full of rows and columns of colored rectangles. MosaicFrame.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.
- [MessageDialog.java](#), a class for displaying modal dialogs that contain a message and one, two, or three buttons. From an example in [Section 7.7](#).
- [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), May 2000

Introduction to Programming Using Java, Third Edition

News and Errata for Version 3.1

NEWS AND ERRATA for Version 3.1 of Introduction to Programming Using Java will be published on the version of the page on the textbook's official site, <http://math.hws.edu/javanotes/>. Here is a link to that page:

<http://math.hws.edu/eck/cs124/javanotes3/errata31.html>

[[Main Index](#)]

Open Publication License

(Draft v1.0, 8 June 1999)

I. REQUIREMENTS ON BOTH UNMODIFIED AND MODIFIED VERSIONS

The Open Publication works may be reproduced and distributed in whole or in part, in any medium physical or electronic, provided that the terms of this license are adhered to, and that this license or an incorporation of it by reference (with any options elected by the author(s) and/or publisher) is displayed in the reproduction.

Proper form for an incorporation by reference is as follows:

Copyright (c) <year> by <author's name or designee>. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, vX.Y or later (the latest version is presently available at <http://www.opencontent.org/openpub/>).

The reference must be immediately followed with any options elected by the author(s) and/or publisher of the document (see section VI).

Commercial redistribution of Open Publication-licensed material is permitted.

Any publication in standard (paper) book form shall require the citation of the original publisher and author. The publisher and author's names shall appear on all outer surfaces of the book. On all outer surfaces of the book the original publisher's name shall be as large as the title of the work and cited as possessive with respect to the title.

II. COPYRIGHT

The copyright to each Open Publication is owned by its author(s) or designee.

III. SCOPE OF LICENSE

The following license terms apply to all Open Publication works, unless otherwise explicitly stated in the document.

Mere aggregation of Open Publication works or a portion of an Open Publication work with other works or programs on the same media shall not cause this license to apply to those other works. The aggregate work shall contain a notice specifying the inclusion of the Open Publication material and appropriate copyright notice.

SEVERABILITY. If any part of this license is found to be unenforceable in any jurisdiction, the remaining portions of the license remain in force.

NO WARRANTY. Open Publication works are licensed and provided "as is" without warranty of any kind, express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose or a warranty of non-infringement.

IV. REQUIREMENTS ON MODIFIED WORKS

All modified versions of documents covered by this license, including translations, anthologies, compilations and partial documents, must meet the following requirements:

1. The modified version must be labeled as such.
2. The person making the modifications must be identified and the modifications dated.
3. Acknowledgement of the original author and publisher if applicable must be retained according to normal

academic citation practices.

4. The location of the original unmodified document must be identified.
5. The original author's (or authors') name(s) may not be used to assert or imply endorsement of the resulting document without the original author's (or authors') permission.

V. GOOD-PRACTICE RECOMMENDATIONS

In addition to the requirements of this license, it is requested from and strongly recommended of redistributors that:

1. If you are distributing Open Publication works on hardcopy or CD-ROM, you provide email notification to the authors of your intent to redistribute at least thirty days before your manuscript or media freeze, to give the authors time to provide updated documents. This notification should describe modifications, if any, made to the document.
2. All substantive modifications (including deletions) be either clearly marked up in the document or else described in an attachment to the document.
3. Finally, while it is not mandatory under this license, it is considered good form to offer a free copy of any hardcopy and CD-ROM expression of an Open Publication-licensed work to its author(s).

VI. LICENSE OPTIONS

The author(s) and/or publisher of an Open Publication-licensed document may elect certain options by appending language to the reference to or copy of the license. These options are considered part of the license instance and must be included with the license (or its incorporation by reference) in derived works.

A. To prohibit distribution of substantively modified versions without the explicit permission of the author(s). "Substantive modification" is defined as a change to the semantic content of the document, and excludes mere changes in format or typographical corrections.

To accomplish this, add the phrase 'Distribution of substantively modified versions of this document is prohibited without the explicit permission of the copyright holder.' to the license reference or copy.

B. To prohibit any publication of this work or derivative works in whole or in part in standard (paper) book form for commercial purposes is prohibited unless prior permission is obtained from the copyright holder.

To accomplish this, add the phrase 'Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.' to the license reference or copy.

OPEN PUBLICATION POLICY APPENDIX:

(This is not considered part of the license.)

Open Publication works are available in source format via the Open Publication home page at <http://works.opencontent.org/>.

Open Publication authors who want to include their own license on Open Publication works may do so, as long as their terms are not more restrictive than the Open Publication license.

If you have questions about the Open Publication License, please contact [David Wiley](#), and/or the Open Publication Authors' List at opal@opencontent.org, via email.

To subscribe to the Open Publication Authors' List:
Send E-mail to opal-request@opencontent.org with the word "subscribe" in the body.

To post to the Open Publication Authors' List:

Send E-mail to opal@opencontent.org or simply reply to a previous post.

To unsubscribe from the Open Publication Authors' List:

Send E-mail to opal-request@opencontent.org with the word "unsubscribe" in the body.